8-2013

The Word Problem for the Automorphism Groups of Right-Angled Artin Groups is in P

Carrie Anne Whittle
University of Arkansas, Fayetteville

The Word Problem for the Automorphism Groups of Right-Angled Artin Groups is in P

The Word Problem for the Automorphism Groups of Right-Angled Artin Groups is in P

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Mathematics

By

Carrie A. Whittle
Northeast Missouri State University
Bachelor of Science in Mathematics and Physics, 1994
Missouri State University
Master of Science in Mathematics, 2008

August 2013
University of Arkansas

This dissertation is approved for recommendation to the Graduate Council.

Dr. Yo'av Rieck
Dissertation Director

Dr. Chaim Goodman-Strauss
Committee Member

Dr. Mark E. Arnold
Committee Member

ABSTRACT

We provide an algorithm which takes any given automorphism ϕ of any given right-angled Artin group G and determines whether or not ϕ is the identity automorphism, thereby solving the word problem for the automorphism groups of right-angled Artin groups. We do this by solving the compressed word problem for right-angled Artin groups, a more general result. A key piece of this solution is the use of Plandowski's algorithm. We also demonstrate that our algorithm runs in polynomial time in the size of the given automorphism, written as a word in Laurence's generators of the automorphism group of the given right-angled Artin group.

ACKNOWLEDGMENTS

I would first like to thank Dr. Yo'av Rieck for the generous amount of time and effort he put into helping me complete this dissertation well. He did an excellent job of guiding my research and teaching me the related concepts along the way, and he went out of his way to assist me in many other aspects of preparing to graduate and finding a job. I greatly appreciate his encouragement and guidance throughout this process.

I would like to gratefully acknowledge Dr. Chaim Goodman-Strauss and Dr. Mark Arnold for taking the time to review my dissertation and for their patience as I completed a last-minute correction. I would also like to thank Dr. Deborah Korth for coordinating my work schedule to accommodate my long commute.

Many professors and instructors have assisted me during my academic career, without whom I would not have reached this point. Among these I would like to thank Mrs. Jane Taylor, from whom I first learned the fun of proofs; Dr. Kevin Easley and Dr. Michael Adams, outstanding professors who made learning math so enjoyable that I became a math major; Dr. Maria DiStefano, who was an enormous encouragement and support to me throughout my undergraduate career; and Dr. Mark Rogers, whose guidance while earning my master's degree helped immensely in preparing me to work on my Ph.D.

My parents, Bruce and Marilyn Willerton, have always encouraged me to pursue my dreams and continue to do so. I am immensely grateful to them for all they have done for me. Likewise I appreciate the many other family members and friends who have supported me during my graduate studies.

I would especially like to express my gratitude to my husband, Brian Whittle, whose willingness to take on even more responsibilities at home enabled me to earn my degree. His understanding, encouragement, and support were critical to my success, and I cannot thank

him enough.

I extend my thanks finally to my daughter, Emma Whittle, whose sparkling personality injects much fun and laughter into my life. She inspires me to be and do my best.

DEDICATION

This dissertation is dedicated to Dr. Michael Jeremy Mallory, whose persistence and cheerfulness in the face of increasingly difficult obstacles was an inspiration to many.

CONTENTS

Introduction **1**

1 Background **3**

1.1 Right-Angled Artin Groups 3

1.2 Straight Line Programs 11

2 Algorithms **13**

2.1 Introduction to the Algorithms 13

2.2 Algorithms, Part 1 – Basics 18

2.3 Algorithms, Part 2 – Preliminaries for Ordering Lexicographically 66

2.4 Algorithms, Part 3 – Lexicographic Ordering 83

2.5 Algorithms, Part 4 – Normal Form 132

Conclusion **143**

Bibliography **168**

INTRODUCTION

Right-angled Artin groups (RAAGs for short) are groups that interpolate between free abelian groups and free groups. They came to the forefront of research in topology in recent years due to the work of Haglund and Wise [3], Agol [1], and others. In this work, they showed that 3-manifold groups are so-called *virtually special* if and only if they are subgroups of right-angled Artin groups, which are also known as graph groups and partially abelian groups. Agol used this to prove the Virtually Haken Conjecture, a very important achievement in topology. Charney has written a nice survey of RAAGs in [2].

We are interested in solving the word problem for the automorphism groups of RAAGs, and we do so by solving the compressed word problem for RAAGs. In [5], Laurence proved that the automorphism group of any right-angled Artin group is finitely generated, and in his proof provided a generating set for the automorphism group of any RAAG. It was shown in [6, 8] that since $\mathrm{Aut}(G)$ is finitely generated for any RAAG G (and since every RAAG is finitely generated), given an automorphism ϕ and an element g of a RAAG, it is possible to construct a *straight line program*, a particular type of compression, which represents $\phi(g)$. In fact, we are able to do this for any finitely generated group, and do it in polynomial time in the number of generators of the automorphism group. We solve the word problem for the automorphism group $\mathrm{Aut}(G)$ of a RAAG G by determining whether or not, for an arbitrary $\phi \in \mathrm{Aut}(G)$, $\phi(a_i) = a_i$ for every a_i in the generating set of G.

Now if we can put an element like $\phi(a_i)$ into a normal form, we can easily tell if it is equivalent to a_i. One way of putting words (that is, elements written as concatenations of generators) from a RAAG into normal form is to put each into its shortest form and order it lexicographically. We refer to this particular normal form as *shortlex* form for short. Hermiller and Meier have shown in [4] that given any word w which represents a given element of a RAAG, there is a process involving only commuting letters and eliminating canceling pairs

of generators which will produce the word w' which is equivalent to w in the group, is in shortest form, and is lexicographically ordered. Since neither of these actions — commuting letters nor eliminating canceling pairs — lengthens the word, the process never increases the length of the word.

However, finding a process which takes a word from a right-angled Artin group and puts it into shortlex form is not trivial, and it is even harder to find a process which runs efficiently. In our solution, we use an inductive process to take two words which are each in shortlex form and find the biggest subwords in the second word that should move into the first word and move them. In this process, each subword is moved as a straight line program; we do not evaluate the words or subwords in G, because that would be extremely inefficient. A key piece that enables our algorithm to run efficiently is Plandowski's algorithm. Plandowski showed in [7] that there is a polynomial-time algorithm which, given two straight line programs, determines whether or not the words produced by those straight line programs are the same. We prove that since each of the two pieces is already shortest and in lexicographic order, there is a constant bound on the number of subwords that move from the second word into the first, and on the number of subwords that move within the second word as a result of moving the subwords which move into the first. This bound enables us to put a word, written as $\phi(g)$ for some $\phi \in \mathrm{Aut}(G)$ and some $g \in G$, and expressed as a straight line program, into shortlex form in polynomial time in the length of the word ϕ written as a concatenation of generators of $\mathrm{Aut}(G)$.

Chapter 1

BACKGROUND

1.1 Right-Angled Artin Groups

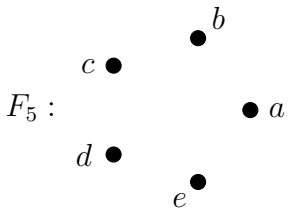
We begin by defining a right-angled Artin group. Perhaps the simplest way to define a right-angled Artin group is by giving its presentation.

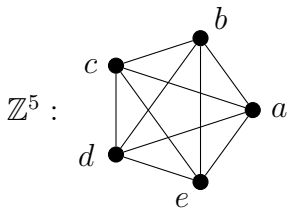
Definition 1.1. A right-angled Artin group has a presentation of the form
$G =< a_1, a_2, \ldots, a_m \mid R >$, where $R \subseteq \{[a_i, a_j] \mid i \neq j\}$.

We see that finitely generated free groups and free abelian groups are each special cases of right-angled Artin groups. However, there are many right-angled Artin groups "in-between" these two classes of groups; we will consider a few examples in a moment. A convenient way to represent a right-angled Artin group is by a simplicial graph, where each vertex corresponds to one generator, and there is an edge between two vertices if and only if the two corresponding generators commute. Let us now look at some specific right-angled Artin groups.

Examples of Right-Angled Artin Groups

The graph representing the free group on n generators is the graph with n vertices and no edges, so the graph to the left below represents F_5. The graph on the right represents $\mathbb{Z}^5$.





The free product of $\mathbb{Z}^2$ with itself is

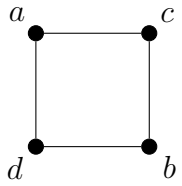
$$\mathbb{Z}^2 * \mathbb{Z}^2 \cong \langle a, b, c, d \mid [a, b], [c, d] \rangle ,$$

and the corresponding graph is pictured to the right.

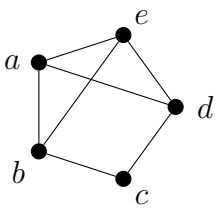
Contrast the previous example with the direct product of the free group on two generators, F_2, with itself, which can be written

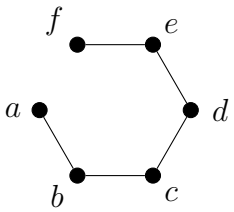
$$F(a, b) \times F(c, d) \cong \langle a, b, c, d \mid [a, c], [a, d], [b, c], [b, d] \rangle ,$$

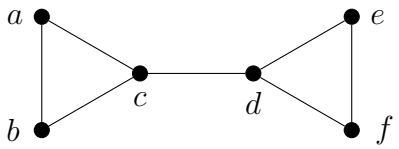
with corresponding graph to the right.



Not all right-angled Artin groups can be decomposed into direct products or free products of free abelian groups and free groups like the examples above. Consider, for example, the right-angled Artin groups corresponding to the graphs below.







Basic Definitions and Notation

It will be helpful to introduce some vocabulary and notation at this point. If A is the generating set of a right-angled Artin group, then by the notation A^{-1} we mean $\{a^{-1} \mid a \in A\}$, the set of inverses of elements of A. We call the elements of the A and A^{-1} *letters*, and the group operation is denoted by concatenation. Naturally, we refer to any finite concatenation of letters and their inverses as a *word*. Thus any word represents an element of the group. The *length* of a word w, denoted $|w|$, is the number of letters it contains.

The empty word — the word with no letters, which we denote by ε — is the identity element. We often use exponents as shorthand for concatenation of a letter with itself; we can write a^3 for the word aaa, for example. When a letter and its inverse are adjacent in a word, we

call this a *canceling pair*, since for any letter a, aa^{-1} is equivalent in the group to the empty word. A word is called *reduced* if it contains no canceling pairs.

Here we must be a bit careful, because there are two ways in which we can speak of two words being the same — they could be identical, letter-for-letter, or they could merely represent the same group element. If we say that two words w_1 and w_2 are *equal*, denoted $w_1 = w_2$, we mean that they are identical, letter-for-letter. If we say that two words are *similar*, denoted $w_1 \simeq w_2$, we mean that they represent the same group element. One immediate consequence of the definition of similar words is that if two words are equal, then they are similar.

The Word Problem

We often want to know if a word is similar to the empty word, and this is called the *word problem* for the group. Since two words w_1 and w_2 are similar if and only if $w_1 (w_2)^{-1} \simeq \varepsilon$, we sometimes think of the word problem as deciding whether or not two words represent the same group element. For free groups and free abelian groups, we can easily find an algorithm that will answer this question.

For free abelian groups, we merely check to see if for each generator a_i, the number of occurrences of a_i in the given word equals the number of occurrences of a_i^{-1} in the word. If so, the word is similar to the empty word, since we can commute all the occurrences of a_i and a_i^{-1} to be next to each other and then cancel them. Otherwise, they are not similar. For example, in $\mathbb{Z}^3$, $abc^{-1}bcb^{-2}a^{-1} \simeq \varepsilon$, but $abc^{-1}b^{-1} \not\simeq \varepsilon$.

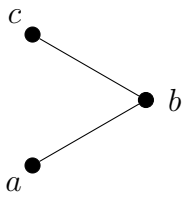
The situation is a bit more complicated with free groups. As with free abelian groups, if for any generator a_i the number of occurrences of a_i in the given word does not equal the number of occurrences of a_i^{-1} in the word, then the word is not equivalent to the empty word in the group. This is true not only for free abelian groups and free groups, but for all right-angled Artin groups. But if the two numbers are equal, the word is not necessarily

the empty word as it was with free abelian groups; consider $aba^{-1}b^{-1}$. So for free groups we remove all canceling pairs, look at the resulting word and remove all canceling pairs from it, etc., until there are no more canceling pairs to remove. At this point, if the resulting word is ε, then the original word was equivalent to the empty word in the group; otherwise it was not. In F_3, for example, $ab^{-1}caa^{-1}c^{-1}ba^{-1} \simeq \varepsilon$, but $ab^{-1}ca^{-1}c^{-1}b \not\simeq \varepsilon$.

Once we allow ourselves to consider right-angled Artin groups which may not be free or free abelian, finding an algorithm is significantly more complicated. We could try doing as we did with free groups — removing canceling pairs repeatedly until there are no more to remove — but even after doing so we may have a word which is not the empty word but which is similar to it.

Examples of Solving the Word Problem

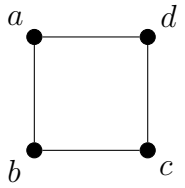
Think about the right-angled Artin group $G =< a, b, c \mid [a, b], [b, c] >$, with corresponding graph to the right, as an example.



After removing canceling pairs from the word $ab^{-1}cc^{-1}b^2a^{-1}c^{-1}b^{-1}c$ in two successive steps, we have the similar word $aba^{-1}c^{-1}b^{-1}c$, which contains no canceling pairs. However, since the letter b commutes with both a and c in this group, we can perform two commutations to arrive at the similar word $aa^{-1}bb^{-1}c^{-1}c$. Removing the three canceling pairs from this word gives us ε.

So now our algorithm involves not only removing canceling pairs, but also commuting letters, and we may have to do each many times. Not only that, but since some pairs of letters may not commute, we must check to see if they do before commuting them, and even then it may not be obvious whether or not it is possible to commute the letters in such a way as to result in more canceling pairs. It would take a fairly long word to illustrate this well, but perhaps

the following short example will hint at this difficulty.



In the group $G =< a, b, c, d \mid [a, b], [a, d], [b, c], [c, d] >$, with corresponding graph to the left, consider the word $bd^{-1}a^{-1}cdb^{-1}c^{-1}a$.

There are no canceling pairs and there is not one single commutation which will result in any canceling pairs; we must perform at least two commutations to get a canceling pair: $bd^{-1}a^{-1}cdb^{-1}c^{-1}a \simeq bd^{-1}a^{-1}dcb^{-1}c^{-1}a \simeq bd^{-1}a^{-1}dcc^{-1}b^{-1}a \simeq bd^{-1}a^{-1}db^{-1}a$. From here, we need only two more commutations and three cancelations to see that this word represents the identity element.

One can imagine how difficult this process could become with many generators and very long words. However, it is also believable that an algorithm exists which would handle this process for any word in any right-angled Artin group. Indeed, it has been shown that the word problem is solvable for right-angled Artin groups. Hermiller and Meier [4] and VanWyk [10] proved that right-angled Artin groups have a *biautomatic structure*. This structure provides a means of solving the word problem.

The Word Problem for Automorphism Groups

We now discuss the word problem for the automorphism group of a right-angled Artin group. As we mentioned in our introduction, in order to solve the word problem for an automorphism group, it is necessary that the automorphism group be finitely generated. That $\mathrm{Aut}(G)$ is finitely generated for any right-angled Artin group G was proven by Laurence [5] and separately by Servatius [9]. Therefore any automorphism of a right-angled Artin group G can be represented by a word whose letters are automorphisms in a fixed generating set. From this point on, we will fix the set of Laurence's generators as our chosen generating set for the automorphism group of any right-angled Artin group. The word problem in this

case is whether or not a given automorphism, say ψ, represents the identity element of the automorphism group. Since the identity element of every automorphism group is the identity map, this involves determining whether or not $\psi(a_i) = a_i$ for every a_i in a generating set of G.

Therefore we know the word problem for the automorphism group is solvable because the word problem for right-angled Artin groups is solvable. However, solving the word problem in the way described above is highly inefficient. Suppose that our given automorphism ψ is the composition of n automorphisms from a chosen finite generating set, say $\psi = \phi_{j_n}\phi_{j_{n-1}}\cdots\phi_{j_1}$. This is a word of length n, but the word in G represented by $\psi(a_1)$, for example, could be exponentially longer. Even in the unlikely case that each of these ϕ_{j_k} takes each generator a_i to a word of only length one or two, the length of the word $\psi(a_i)$ written in generators of G could be about $(3/2)^n$. For example, suppose a and b are two elements of a generating set for G and that $\phi : G \to G$ is partially given by $\phi(a) = ab$ and $\phi(b) = a$. Now if $\psi = \phi^6$, a word of 6 letters in $\mathrm{Aut}(G)$, then

$$\begin{aligned}
\psi(a) &= \phi^6(a) \\
&= \phi^5(ab) \\
&= \phi^4(aba) \\
&= \phi^3(abaab) \\
&= \phi^2(abaababa) \\
&= \phi(abaababaabaab) \\
&= abaababaabaababaababa,
\end{aligned}$$

a word of 21 letters in G.

This illustrates how the automorphism group of a right-angled Artin group G can be used to *compress* words in G. One may notice that this compression only shortens words in G

which have repetitions or patterns, as in the example above. However, most of the strings that actually occur in our work have this characteristic.

Since the length of the word in G represented by a word in $\mathrm{Aut}(G)$ is, in general, exponentially longer than the word in $\mathrm{Aut}(G)$, using any solution for the word problem in G which requires evaluating each $\psi(a_i)$ to solve the word problem in $\mathrm{Aut}(G)$ will be extremely slow and take at least exponentially long time. In our research we give a polynomial-time solution to the word problem for $\mathrm{Aut}(G)$. We actually give a more general result — a solution to the compressed word problem for G which is polynomial in the size of the compression. This result implies a polynomial-time solution to the word problem for $\mathrm{Aut}(G)$, since the automorphism group provides one kind of compression.

Further Definitions

Before we can discuss the algorithms which will provide a solution to the compressed word problem for RAAGs, we must introduce more vocabulary. The following definitions all occur in the context of a right-angled Artin group $G =< a_1, a_2, \ldots, a_m \mid R >$, where $R \subseteq \{[a_i, a_j] \mid i \neq j\}$.

We say a_i *blocks* a_j if $[a_i, a_j]$ is not a relator of G. Given an ordering on a set of letters, we say the letter a is *lighter* than the letter b, and that b is *heavier* than a, if a comes before b in the ordering. We denote this by $a < b$ or $b > a$. We say a_i *hinders* a_j on the left (right), denoted $a_i|a_j$ ($a_j|a_i$), if $i \leq j$ ($j \leq i$) or $[a_i, a_j]$ is not in R.

We will often use abbreviations for specific subwords of a given word. For a fixed word w with $0 \leq i \leq j \leq |w|$, $w[i:j]$ represents the string of characters beginning immediately after the i^{th} character and ending immediately after the j^{th} character. So for the word $w = abcd$, $w[1:3] = bc$ and $w[2:2] = \varepsilon$. By using a negative index, we indicate counting from the end; hence $w[1,-1] = w[1,|w|-1] = w[1:3] = bc$. We use $w[i]$ to indicate $w[i:i+1]$, the

$i + 1^{\text{th}}$ character; $w[: j]$ to indicate $w[0 : j]$, the leftmost subword of length j; and $w[i :]$ to indicate $w[i : |w|]$, the rightmost subword of length $|w| - i$. Thus $w[1] = b$, $w[: 1] = a$, and $w[1 :] = bcd$; furthermore, $w[0] = a$ and $w[-1] = d$.

Let G be a group with an ordering on its generators, and let w be a word in the generators of G. Then w is said to be in *lexicographic order* if for any other word w' which is similar to w and contains the same letters as w the following condition holds: If i is the smallest integer such that $w[i] \neq w'[i]$, then $w[i]$ comes before $w'[i]$ in the ordering given. Lemma 1.2 follows from this definition.

Lemma 1.2. *A word w representing an element of a right-angled Artin group is in lexicographic order if and only if for any letter $w[i]$ in w, all letters between $w[i]$ and the rightmost letter left of $w[i]$ with which $w[i]$ does not commute come before $w[i]$ in the lexicographic ordering.*

Proof. We show sufficiency by proving the contrapositive. Suppose that in the word w there are letters $w[g]$, $w[h]$, and $w[i]$ in w such that $w[g]$ is the rightmost letter left of $w[i]$ not commuting with $w[i]$, that $w[h]$ lies between $w[g]$ and $w[i]$, and that $w[h]$ is heavier than $w[i]$. Without loss of generality, assume that $w[h]$ is the leftmost letter between $w[g]$ and $w[i]$ which is heavier than $w[i]$. Then the word $v = w[: h - 1] \cdot w[i] \cdot w[h : i - 1] \cdot w[i :]$ is similar to w since by the choice of $w[g]$, $w[i]$ commutes with all letters between $w[g]$ and $w[i]$. (Recall that $w[h] = w[h - 1 : h]$.) Furthermore, $v[: h - 1] = w[: h - 1]$, and the h^{th} letter of v is $w[i]$, which is lighter than $w[h]$. Thus the smallest integer j for which $w[j] \neq v[j]$ is h, and $v[h] < w[h]$. Hence w is not in lexicographic order.

We also prove the contrapositive to show necessity. Let w be a word which is not in lexicographic order, and let v be similar to w and have the same letters as w but be in lexicographic order. Let h be the smallest integer for which $v[h] \neq w[h]$; then by definition, $v[h] < w[h]$. Now $v[: h - 1] = w[: h - 1]$, so the letter $v[h]$ must occur in $w[h :]$; let $w[i]$ be the leftmost occurrence of $v[h]$ in $w[h :]$. Since $v \simeq w$, $w[i]$ must commute with every letter

in $w[h : i-1]$, so the rightmost letter left of $w[i]$ not commuting with $w[i]$ lies in $w[: h-1]$. Moreover, $w[i] = v[h] < w[h]$, so the letter $w[h]$ is heavier than $w[i]$ and lies between $w[i]$ and the rightmost letter left of $w[i]$ which does not commute with $w[i]$. □

1.2 Straight Line Programs

The tool that we use to represent a given automorphism is a straight line program. These, and related terms, are defined below.

A *straight line program* $\mathbb{A} =< \mathcal{L}, \mathcal{A}, A_n, \mathcal{P} >$ consists of the following: a finite set $\mathcal{L} = \{a_1, a_2, \ldots, a_m\}$ of letters, called *terminal* characters; a disjoint finite set $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ of *non-terminal* characters; a *root* non-terminal A_n; and a set $\mathcal{P} = \{A_i \to Q_i\}$ of *production rules*. Each production rule causes a non-terminal A_i to be replaced with its *production*, Q_i, which is a word whose letters are all in $\mathcal{L} \cup \mathcal{A}$. For any nonterminal A_j appearing in Q_i, it must be the case that $j < i$.

To run the straight line program $\mathbb{A}$, we replace the root A_n with its production Q_n, then replace the non-terminals in the production with their productions, and so forth until only terminal characters remain in the word that results. We denote this word by $w(A)$ or w_A; similarly the word produced by the non-terminal A_i is denoted $w(A_i)$ or w_{A_i}.

We can illustrate this process with a *production tree*. Let the tree for a terminal character a_i be a vertex labeled a_i, and let the tree for a non-terminal A_i be a planar graph with a vertex labeled A_i connected by edges to a copy of the tree for each character in its production Q_i, where the trees for each character appear in order from left to right. The vertices labeled with terminal characters are often called *leaves*. The word formed by the labels of the leaves of the production tree for $\mathbb{A}$ is the word $w(A)$ produced by the straight line program.

A straight line program is said to be in *Chomsky normal form* if all of the productions Q_i

have length one or two, where the productions of length one are terminal characters and the productions of length two consist of two nonterminal characters.

Chapter 2

ALGORITHMS

2.1 Introduction to the Algorithms

Before discussing the algorithms in detail, it will be helpful to comment about some differences between our routines and those for standard straight line programs. We wrote our routines in Python, and the entire Python program which puts a word into shortlex form is included in the appendix.

First we should note that instead of using straight line programs as defined in Section 1.2, we created an object type called SLP which we use instead, along with the global constant m, which is the number of generators of the right-angled Artin group. The attributes of the object SLP are **P**, which is the list holding the production rules of the associated straight line program; $normform$, which is set to true if the SLP object is in quasinormal form (which we explain below) and set to false if not; **gens**, a list of length m with each item **gens**$[i]$ set to the string 'ai'; **invgens**, a list of length m with each item **invgens**$[i]$ set to the string 'Ai'; and $numbr$, which is set to the length of the list **P**.

It is straightforward to convert between a straight line program in Chomsky normal form and an SLP object. Given a straight line program $\mathbb{B} =< \mathcal{L}, \mathcal{B}, B, \mathcal{P} >$, we let m be the number of elements of $\mathcal{L}$, let **P** be the list of ProdRule objects (also discussed below) associated with $\mathcal{P}$, let **gens** and **invgens** be the lists described above, and let $numbr$ be the length of the list **P**. On the other hand, given m and an SLP $\mathbb{B}$, we set $\mathcal{L}$ to be $\{a_0, a_1, \ldots, a_{m-1}\}$, set $\mathcal{P}$ to be the set of production rules indicated by the list **P**, set $\mathcal{B}$ to be the set of all non-terminal characters that appear in $\mathcal{P}$, and let $B = B_{numbr-1}$. Notice that by letting $B = B_{numbr-1}$ we are setting the root to B_k, where k is the largest index of any non-terminal character. It

is standard to index the terminal and non-terminal characters from 1 to *numbr*. However, Python indexes its lists starting with 0, so we index the terminal characters from 0 to $m-1$ and the non-terminal characters from 0 to $numbr-1$. Thus the root terminal has index $numbr-1$ rather than *numbr*. Note that changing the indices can be done in linear time, so we need not worry about this detail when showing that the programs run in polynomial time. Furthermore, this process of converting from a straight line program to an SLP object or vice-versa also runs in linear time in *numbr*. We state this formally below.

Lemma 2.1. *An algorithm exists which, given a straight line program $\mathbb{A}$ in Chomsky normal form, produces an SLP object $\mathbb{B}$ such that $w_B = w_A$, and which runs in polynomial time in the length of $\mathbb{A}$. Similarly, an algorithm exists which, given an SLP $\mathbb{B}$, produces a straight line program $\mathbb{A}$ in Chomsky normal form such that $w_A = w_B$, and which runs in polynomial time in the number of ProdRules in $\mathbb{B}$.*

We now explain how we store production rules as ProdRule objects in the list **P**. For straight line programs in Chomsky normal form, all productions of length one are terminal characters; these production rules are of the form $B_i \to a_r^{\pm 1}$. In our programs, we store this type of production rule as a string in the i^{th} item of the appropriate list, where that string is 'ar' if $B_i \to a_r$ or 'Ar' if $B_i \to a_r^{-1}$. We should at this point clarify our use of the notation. In the Python programs, we use '$a0$' and '$A0$', for example, to represent the letter a_0 and its inverse, respectively. In the proofs, we often call the non-terminal characters A_j, so A_0 is the first non-terminal character. When using the notation 'Ar' in close proximity to the discussion of non-terminal characters, we will use B_j rather than A_j for the non-terminal characters to help avoid confusion.

All productions of length two in Chomsky normal form are the concatenation of two non-terminal characters; these production rules are of the form $B_i \to B_j \cdot B_k$, where $j < i$ and $k < i$. In our routines we created a new object type called Concat to store this type of production rule. An object of type Concat has attributes $idx1$ and $idx2$, and for the rule

$B_i \to B_j \cdot B_k$, we store j in $idx1$ and k in $idx2$ of a Concat object, which is stored in item i of the appropriate list. We found the need to create another object type, Pr1Pr, to store production rules whose productions have length one but are non-terminal characters. (These are not included in Chomsky normal form.) An object of type Pr1Pr has an attribute $idx1$ where we store j for a rule of the form $B_i \to B_j$; this Pr1Pr object is then stored in item i of the corresponding list. We also created an object type called ProdRule to encompass all three of the types of production rules described above; the attribute wi of a ProdRule object is either a string, a Concat object, or a Pr1Pr object, depending on the type of production it describes. Technically, all of the Concat and Pr1Pr objects and all strings used as terminal characters are stored in the wi attribute of a ProdRule object before being put into their lists. We will refer to both actual production rules, as defined above, and ProdRule objects as 'production rules'; which of the two is being referred to will be clear from the context. We should note that a straight line program which is not in Chomsky normal form can have production rules with productions consisting of more than two characters, but there is no ProdRule object in our routines which can store such a production rule, so no SLP object has any such production rules.

As indicated above, we use a slightly different normal form than Chomsky normal form for the straight line programs in our routines. To differentiate, we will use "normal form" to refer to Chomsky normal form and "quasinormal form" to refer to the normal form we use. We explain quasinormal form in the next paragraph.

One way in which the two forms are different is that a straight line program in normal form has no productions of length one which are non-terminal characters. In quasinormal form we allow this type of production, but only when the word produced by the straight line program is a single letter, for reasons we explain presently. For every terminal character in a straight line program in our Python routines, there is a production rule whose production is that terminal character; this is not required in normal form. This does not mean, however, that every

terminal character appears in every word produced by a straight line program in our routines; each letter only appears in the produced word if some non-terminal character has a production rule pointing to the non-terminal character which points to that letter. For example, if $\mathcal{P} = \{B_0 \to a_0, B_1 \to a_1, B_2 \to a_2, B_3 \to a_0^{-1}, B_4 \to a_1^{-1}, B_5 \to a_2^{-1}, B_6 \to B_4 \cdot B_0\}$, then the straight line program in our routines having this set of production rules would produce the word $a_1^{-1}a_0$. Similarly, the straight line program having the set of production rules $\mathcal{P} = \{B_0 \to a_0, B_1 \to a_1, B_2 \to a_2, B_3 \to a_0^{-1}, B_4 \to a_1^{-1}, B_5 \to a_2^{-1}\}$ would produce the empty word. To produce a single letter, we must have a production rule pointing to the non-terminal character which points to that letter. For example, the program with the set of rules $\mathcal{P} = \{B_0 \to a_0, B_1 \to a_1, B_2 \to a_2, B_3 \to a_0^{-1}, B_4 \to a_1^{-1}, B_5 \to a_2^{-1}, B_6 \to B_2\}$ produces the word a_2.

In quasinormal form we require the rules whose productions are terminal characters to have smaller indices than those which produce non-terminal characters. So while a straight line program with the set of rules $\mathcal{P} = \{B_0 \to a_0, B_1 \to a_1, B_2 \to B_0 \cdot B_1, B_3 \to a_2, B_4 \to a_0^{-1}, B_5 \to a_1^{-1}, B_6 \to a_2^{-1}, B_7 \to B_4 \cdot B_2\}$ is in normal form, it is not in quasinormal form. A similarity between normal form and quasinormal form is that the empty word is not allowed as a production in either form. A formal definition of quasinormal form is given below.

Definition 2.2. An SLP object $\mathbb{A} =< \mathbf{P} >$ is said to be in *quasinormal form* if the following conditions are met:

1. For each item **gens**$[i]$ in the list **gens** and each item **invgens**$[j]$ in **invgens**, **P** contains one string-type ProdRule set to **gens**$[i]$ or **invgens**$[j]$, respectively;

2. All string-type ProdRules have smaller indices in **P** than Pr1Pr- or Concat-type ProdRules;

3. The empty letter does not occur in **P**; and

4. There is no more than one Pr1Pr-type ProdRule in **P**, and if **P** contains a Pr1Pr-type

ProdRule, the word produced by $\mathbb{A}$ is a single letter.

Due to limited space in the flowcharts, the names of some variables in the Python programs have been shortened in the flowcharts. However, the shorter names were chosen so that there should be no confusion as to which variable each corresponds to. Additionally, some of the names of the programs are longer and more descriptive in the flowcharts, but it should be obvious which flowchart corresponds with which Python routine.

At the beginning of many of the routines, the variable n is set to the number of non-terminal characters, the variable **thisP** is set to the list containing the productions, and/or other variables are set to particular attributes of the SLP. These are not indicated in the flowcharts because of the limited space there, and because any such steps are indicated in the information about what is input into each routine at the beginning. Additionally, any such steps run in linear time, so we may ignore them in our proofs that the routines run in linear or polynomial time. In some of the routines, there are conditions checked which, depending on whether or not the condition is true, cause the routine to determine the value to return with very little work and then immediately exit. These are not indicated in the flowcharts because of the space they would take up, but they are discussed in the proofs.

Remark 2.3. Some of the operations which occur in our algorithms, such as determining whether a given value is less than n or creating a list of length n, do not actually occur in constant time as claimed in the lemmas below; the time required does depend on n. However, the time required for these operations is on the order of $n \log n$, so claiming that the time required is bounded by a constant does not change the results of the proofs that the algorithms run in polynomial time.

For each algorithm involved in solving the word problem for a given automorphism of a right-angled Artin group, we include a lemma and proof that the algorithm does what it is supposed to and a separate lemma and proof that the algorithm runs in polynomial (or linear) time. We begin by considering routines which stand alone and progress to the one

which uses all of the others (some indirectly) and which puts the word produced by a given SLP into shortest form and lexicographic order. All algorithms which call other routines appear after the routines they call in this paper.

2.2 Algorithms, Part 1 – Basics

We begin with two fairly simple routines which give us basic information, namely the length of the produced word and which generators appear in the produced word, about the SLP which is input.

Lemma 2.4. *The algorithm Get Length described by the flowchart in Figure 2.1 returns the length of the word produced by the SLP which is input.*

Proof. We begin by creating a list called **prL** of length n, the number of production rules in $\mathbb{A}$, the SLP being input, setting each item in the list to 0. Each item in **prL** corresponds to a non-terminal character, so item 0 in the list corresponds to A_0, item 1 corresponds to A_1, and so on. We proceed through the production rules of $\mathbb{A}$, considering one at a time, as the index i steps from 0 to $n-1$. Since $\mathbb{A}$ is in quasinormal form, the first production rules encountered are those pointing to terminal characters, and for each of these, the corresponding item in **prL** is set to 1. If at the i^{th} step the production rule is $A_i \to A_r$, then $\mathbf{prL}[i]$ is set to $\mathbf{prL}[r]$. All other production rules must be of the form $A_i \to A_r \cdot A_s$, in which case we set $\mathbf{prL}[i]$ to $\mathbf{prL}[r] + \mathbf{prL}[s]$. Since $i > r, s$ for any such production rule (and $i > r$ in the previous case), the values of $\mathbf{prL}[r]$ and, if applicable, $\mathbf{prL}[s]$ have already been calculated. Thus at the end of each step of the process, $\mathbf{prL}[i]$ will contain the length of $w(A_i)$. Therefore the last item in **prL** will contain the length of w_A, which is the word produced by $\mathbb{A}$. This is the number which the algorithm returns. □

Lemma 2.5. *The algorithm Get Length described by the flowchart in Figure 2.1 runs in polynomial time in n, the length of the SLP which is input.*

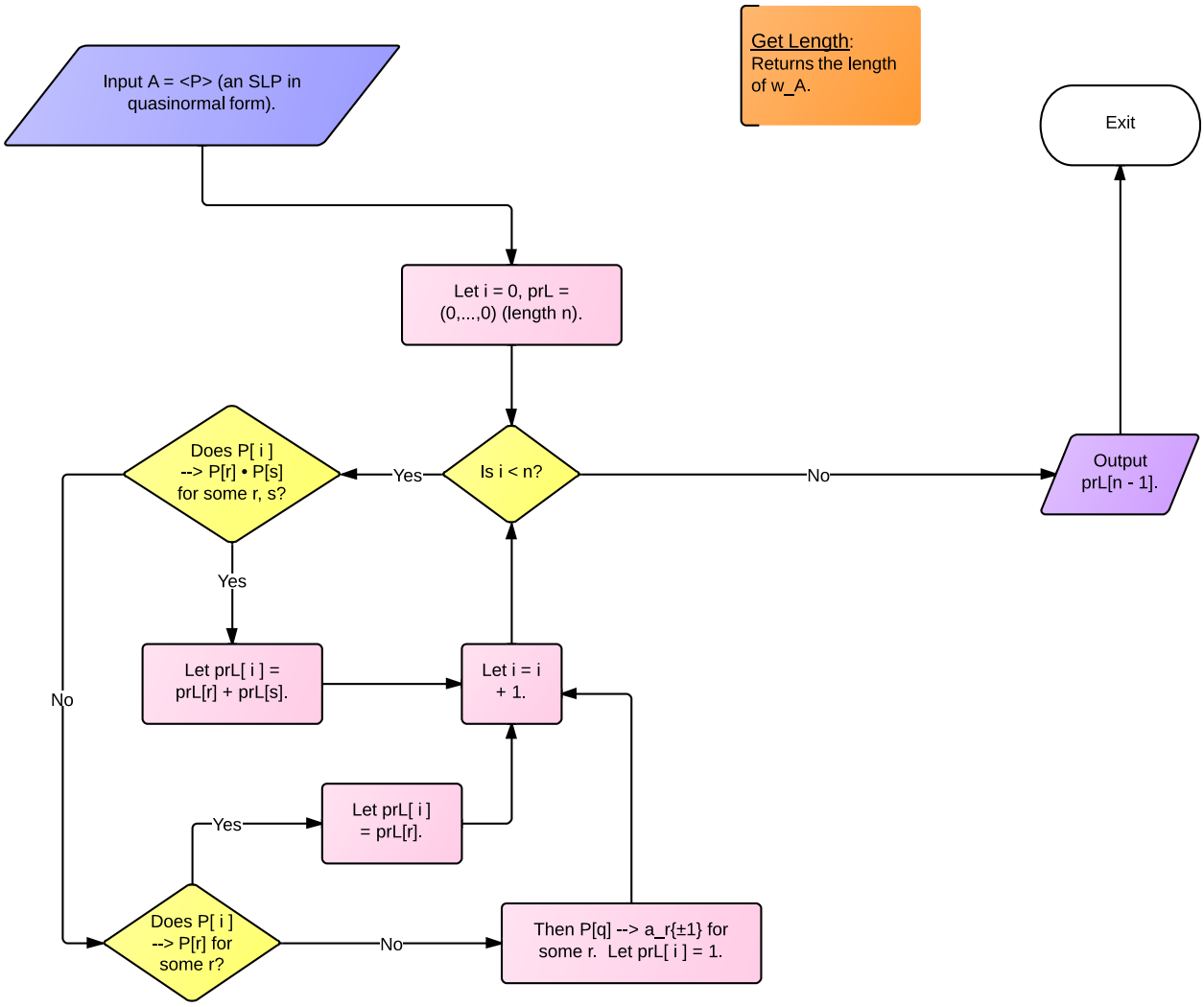


Figure 2.1: Algorithm Get Length

Proof. The time it takes to run the operations that happen outside the main loop, as well as the time required to determine whether or not $i < n$, do not depend on n and so happen in constant time, say c_1 steps. For each iteration of the main loop, the time is also independent of n and so the number of steps is bounded by a constant, say c_2. The loop runs n times, so the total number of steps is no more than $c_2 n + c_1$. The actual time required for Get Length is on the order of $n \log n$ by Remark 2.3, so the algorithm runs in polynomial time. □

Lemma 2.6. *The algorithm Find Included Generators described by the flowchart in Figure 2.2 returns a list* $\mathbf{z}$ *of length* m *where each item* $\mathbf{z}[i]$ *is 1 if the generator* a_i *or its inverse occurs in the word produced by the SLP which is input and is 0 otherwise.*

Proof. We input an SLP $\mathbb{A} =< \mathcal{L}, \mathcal{A}, A, \mathcal{P} >$, and we create a list, $\mathbf{z}$, of length m, setting each list item to 0. In the main loop, we consider one generator at a time, a_i. During the i^{th} step through the loop we begin by initializing another list, $\mathbf{z1}$, containing n items all set to 0. Each item in $\mathbf{z1}$ corresponds to a non-terminal character, and we use the index j to step through the production rules of $\mathbb{A}$ one at a time, starting with $j = 0$. Since $\mathbb{A}$ is in quasinormal form, the first production rules encountered are those pointing to terminal characters. For these, if the production rule is $A_j \to a_i$ or $A_j \to a_i^{-1}$, then item $\mathbf{z1}[j]$ is set to 1; otherwise $\mathbf{z1}[j]$ is left unchanged. If the production rule is $A_j \to A_r$, then $\mathbf{z1}[j]$ is set to 1 if and only if $\mathbf{z1}[r] = 1$; otherwise it is not changed. All other production rules must be of the form $A_j \to A_r \cdot A_s$. In this case, if $\mathbf{z1}[r] = 1$ or $\mathbf{z1}[s] = 1$, $\mathbf{z1}[j]$ is set to 1; otherwise it remains 0. Since $j > r, s$ for any such production rule (and $j > r$ in the previous case), the values of $\mathbf{z1}[r]$ and, if applicable, $\mathbf{z1}[s]$ have already been calculated. So once j has progressed through all of the production rules, the last item in $\mathbf{z1}$, that is, $\mathbf{z1}[n-1]$, will be 1 if and only if a_i or a_i^{-1} occurs in w_X. Having run through the inner loop n times, we find ourselves at the end of the i^{th} step of the main loop. We now change $\mathbf{z}[i]$ to 1 if $\mathbf{z1}[n-1] = 1$; otherwise it remains 0. Thus after stepping through all m steps of the main loop, we see that for each i, $\mathbf{z}[i] = 1$ if and only if $a_i^{\pm 1}$ occurs in w_A, and all items that are not 1 are 0. □

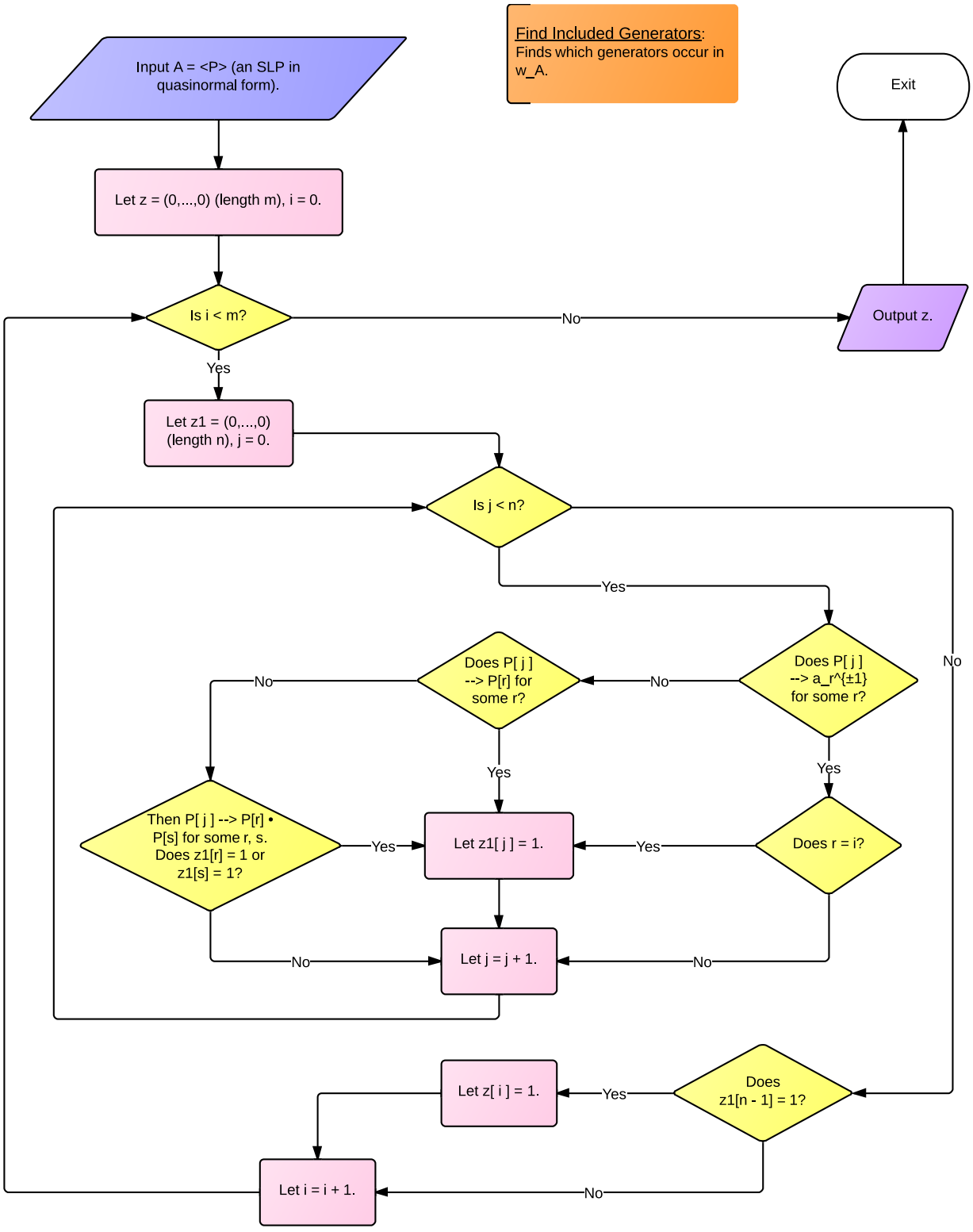


Figure 2.2: Algorithm Find Included Generators

Lemma 2.7. *The algorithm Find Included Generators described by the flowchart in Figure 2.2 runs in polynomial time in n, the length of the SLP which is input.*

Proof. The time it takes to run the operations that happen outside the main loop, as well as the time required to determine whether or not $j < m$, do not depend on n and so happen in constant time, say c_1 steps. The main loop is run m times, and m is a constant for a given right-angled Artin group. For each iteration of the smaller loop, the time is independent of n, and so the number of steps is bounded by a constant, say c_2. The smaller loop runs n times, so the total number of steps is no more than $c_2 \cdot n \cdot m + c_1 = c_3 n + c_1$. The number of steps is actually on the order of $n \log n$ because of Remark 2.3, so Find Included Generators runs in polynomial time. □

The routine Letter to SLP described below is used only in the Put In Lexicographic Order and Make It Shortest algorithms. In these, we need to create an SLP which produces a single given letter; that is what this algorithm does.

Lemma 2.8. *The algorithm Letter to SLP described by the flowchart in Figure 2.3 inputs a ProdRule R. If R is of the form $R \to a_r^{\pm 1}$ for some r, Letter to SLP returns an SLP in quasinormal form which produces the letter $a_r^{\pm 1}$; otherwise Letter to SLP returns an SLP which produces ε.*

Proof. After we input a ProdRule R, we create an empty list $\mathbf{Q}$; let $\mathbb{B}$ be $< \mathbf{Q} >$, the SLP having $\mathbf{Q}$ as its set of production rules; and let $s = R$. We check to see if $s \to a_r^{\pm 1}$ for some r, and if not, we return the SLP $\mathbb{B}$, which at this point has an empty list of production rules and therefore produces ε. Otherwise we continue with the rest of the routine. Each item in $\mathbf{Q}$ will correspond to a non-terminal character. We use the index i to step twice through the values $0, 1, \ldots, m-1$, beginning with $i = 0$. In the first loop, we first check to see if $s \to a_i$, and if so, we let the variable GI equal i. Whether or not $s \to a_i$, we append the rule $\mathbf{Q}[i] \to a_i$ to $\mathbf{Q}$, indicating the new rule $B_i \to a_i$. We then return to the beginning of the loop. The

first loop, therefore, adds the ProdRules corresponding to the non-terminal characters which point to the generators (but not their inverses) to the list $\mathbf{Q}$, and if s points to one of these a_i, then GI is set to that i. The second loop is similar to the first, the difference being that we are concerned with the inverses of the generators this time. In the second loop, if $s \to a_i^{-1}$, we let the variable GI equal $i + m$. In either case, we append the rule $\mathbf{Q}[i + m] \to a_i^{-1}$ to $\mathbf{Q}$, indicating the new rule $B_{i+m} \to a_i^{-1}$. We then return to the beginning of the loop. So the second loop adds the ProdRules corresponding to the non-terminal characters which point to the inverses of the generators to the list $\mathbf{Q}$, and if s points to one of these a_i^{-1}, then GI is set to that i plus m. Finally we append the production rule $\mathbf{Q}[2m] \to \mathbf{Q}[GI]$ to $\mathbf{Q}$, indicating the rule $B_{2m} \to B_{GI}$. We create a new SLP $\mathbb{B}$ with this new set of production rules. Now, therefore, the root non-terminal is B_{2m}, and the production rule for B_{2m} is $B_{2m} \to B_{GI}$. Also, by construction, the rule for B_{GI} is $B_{GI} = B_r =\!\!\to a_r$ if $s \to a_r$ and is $B_{GI} = B_{r+m} \to a_r^{-1}$ if $s \to a_r^{-1}$. Therefore if $s \to a_r^{\pm 1}$ for some r, the SLP $\mathbb{B}$ produces $a_r^{\pm 1}$. It is straightforward to check that $\mathbb{B}$ is in quasinormal form. □

Lemma 2.9. *The algorithm Letter to SLP described by the flowchart in Figure 2.3 runs in constant time.*

Proof. None of the steps in Letter to SLP are dependent on the length of any SLP, so the algorithm runs in constant time. □

The following algorithm is much more complicated than the previous ones we have considered. In fact, the flowchart would not fit on a single page, which is why it is broken into two flowcharts. Since quasinormalizing an SLP occurs as part of most of the following routines, it is necessary to address Quasinormalize SLP early in our discussion of the algorithms.

Lemma 2.10. *The algorithm Quasinormalize SLP described by the flowcharts in Figures 2.4 and 2.5 returns an SLP in quasinormal form which produces the same word as that produced by the SLP which is input.*

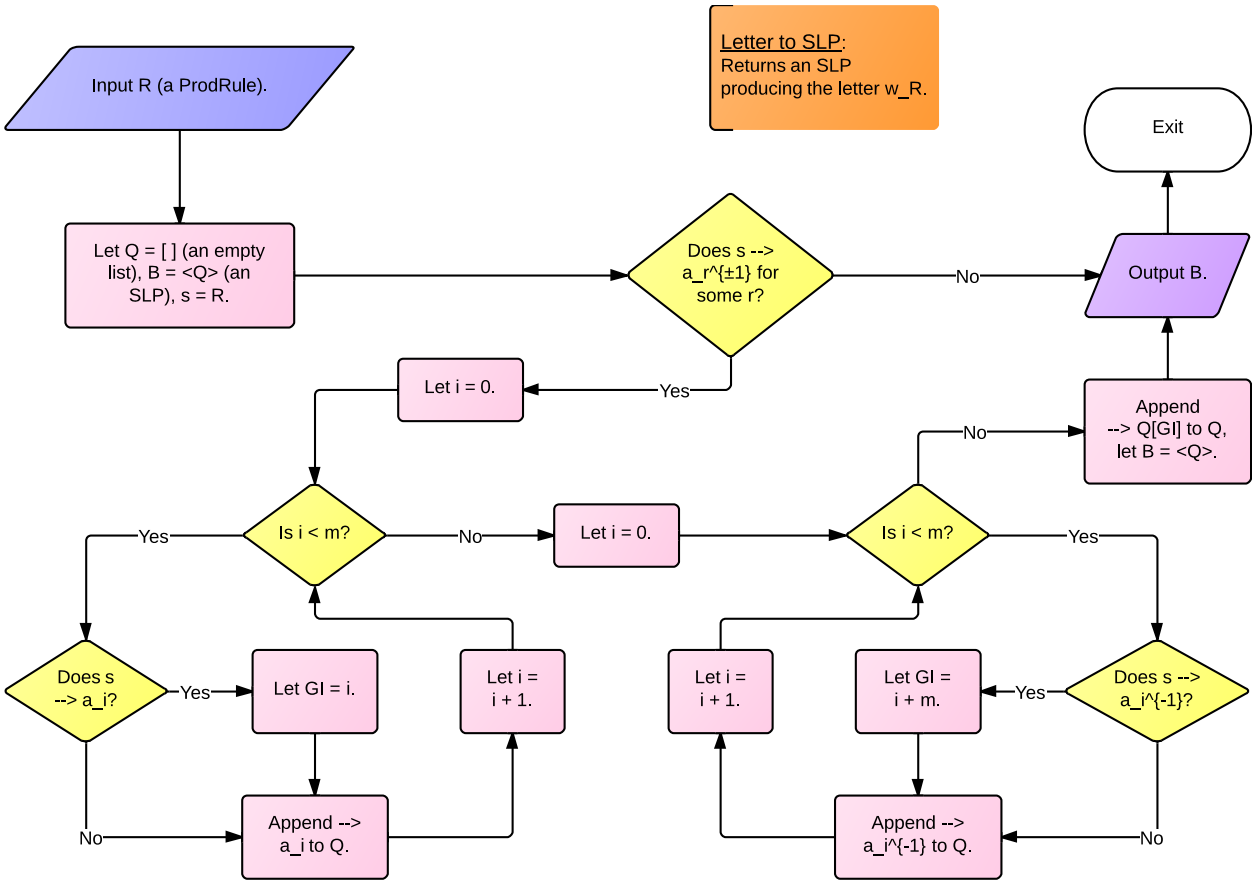


Figure 2.3: Algorithm Letter to SLP

Proof. The reader may find it helpful to refer to Example 2.13 while proceeding through this proof. After we input the SLP $\mathbb{A}$ with list $\mathbf{P}$ of production rules, we begin by creating two empty lists, $\mathbf{Q}$ and $\mathbf{L}$, and two lists of length n, $\mathbf{z}$ and $\mathbf{z2}$. We set each item of $\mathbf{z}$ to 0 and each item of $\mathbf{z2}$ to -1. We also initialize the variables q and oL to 0 and true, respectively. Let us discuss the role each of these variables plays.

We will use $\mathbf{Q}$ to hold the new production rules. The list $\mathbf{L}$ will allow us to keep track of which terminal characters we have already encountered so that we do not repeat any, and the index of a terminal character in $\mathbf{L}$ will be the index of the corresponding production in $\mathbf{Q}$. For each i, item i of $\mathbf{z}$ will be the difference between i and the index in $\mathbf{Q}$ of the production that corresponds to $\mathbf{P}[i]$, that is, the production of A_i. (Recall that for each i, the ProdRule in $\mathbf{P}i$ is the production of A_i.) For each i such that $\mathbf{P}[i]$ is a terminal character, the i^{th} item of $\mathbf{z2}$ will be the index in $\mathbf{Q}$ of the terminal character $\mathbf{P}[i]$; for any other i, $\mathbf{z2}[i]$ will remain -1. The variable q is used to keep track of the index of the current item of $\mathbf{Q}$. The variable name oL is short for 'only letters'; oL is set to true at the beginning and is changed to false when and only when a production containing a non-terminal character, that is, a Concat- or Pr1Pr-type ProdRule, which produces a word of length at least 1 is encountered. Thus if oL is true at the end of the program, an SLP with empty production list $\mathbf{Q}$ is returned.

In the first small loop we use the index i to step through the production rules of $\mathbb{A}$ one at a time, beginning with $i = 0$. In this loop, we are concerned only with those production rules of $\mathbb{A}$ whose productions are terminal characters, so we first check to see if the current rule is $A_i \to a_r^{\pm 1}$ for some r; if not, we proceed to the next i. (If the rule is $A_i \to \varepsilon$, we also proceed to the next i.) If the rule is $A_i \to a_r^{\pm 1}$ for some $r \in \{0, 1, \ldots, m-1\}$, we check to see if $a_r^{\pm 1}$ is in the list $\mathbf{L}$. If not, we append $a_r^{\pm 1}$ to $\mathbf{L}$ and also append $a_r^{\pm 1}$ to $\mathbf{Q}$, indicating the new rule $B_q \to a_r^{\pm 1}$, and we increase the index q by 1. Whether $a_r^{\pm 1}$ is already in $\mathbf{L}$ or not, we change $\mathbf{z2}[i]$ to be the index of $a_r^{\pm 1}$ in $\mathbf{L}$. We then proceed to the next i. We exit this loop when $i = n$.

After exiting this first small loop, we proceed to the main loop. (This is the part of the routine illustrated in the flowchart in Figure 2.5.) As in the previous loop, we will use the index i to step through the production rules of $\mathbb{A}$ one at a time, beginning with $i = 0$. Before beginning to consider the production rules, we first create two empty lists, $\mathbf{l}$ and $\mathbf{e}$; initialize the variables lC and lP each to -1 and the variable k to 0; and create a list $\mathbf{CL}$ of length n with each item set to 0. We use $\mathbf{l}$ to keep track of which letters have been encountered in the productions before the current step of the loop. At each step, the list $\mathbf{e}$ will contain the indices of those non-terminal characters previously encountered which produce the empty word. The variable k holds the number of production rules in $\mathbb{A}$ already encountered by that step for which a new corresponding production in $\mathbf{Q}$ is not created (as with non-terminal characters which produce the empty letter or with Pr1Pr-type ProdRules). $\mathbf{CL}$ keeps track of which items in $\mathbf{Q}$ contain Concat-type ProdRules. The variable name lC is short for 'last Concat'; this variable contains the highest index in $\mathbf{P}$ of a Concat-type ProdRule whose corresponding production rule in $\mathbf{Q}$ is a Concat ProdRule. Similarly, lP is short for 'last Pr1Pr', and this variable contains the highest index in $\mathbf{P}$ of either a Pr1Pr-type ProdRule whose production is not the empty word or a Concat-type ProdRule with one of the two non-terminal characters in the production producing the empty word. We will see the need for $\mathbf{CL}$, lC, and lP when we discuss the end of the algorithm.

As we encounter each production rule of $\mathbb{A}$ during the main loop, there are three possibilities. The first is that the rule is of the form $A_i \to A_r$ for some r. In this case, we do not add a ProdRule to $\mathbf{Q}$, because it would need to be of the Pr1Pr type, and those are not allowed in quasinormal form except when the word produced by the SLP is a single letter. Instead, we want to replace any occurrence of A_i with A_r, eliminating the need for A_i. Of course, we make no changes to $\mathbb{A}$ but instead effect this in our new set of rules in $\mathbf{Q}$. To achieve this, we set $\mathbf{z}[i]$ to be $i - r + \mathbf{z}[r]$ because $r - \mathbf{z}[r]$ is the index in $\mathbf{Q}$ of the production that corresponds to the production of A_i. (Recall that for each j already encountered, $\mathbf{z}[j]$ is the difference between j and the index in $\mathbf{Q}$ of the production corresponding to that of A_j.)

Letting $\mathbf{z}[i] = i - r + \mathbf{z}[r] = i - (r - \mathbf{z}[r])$ thus causes $\mathbf{z}[i]$ to be the difference between i and the index in $\mathbf{Q}$ of the production corresponding to that of A_r so that the production in $\mathbf{Q}$ corresponding to that of A_i is the production in $\mathbf{Q}$ corresponding to that of A_r. Because we are not adding a new rule to $\mathbf{Q}$ at this step, we also increase the value of k by 1. We then check to see if r occurs in the list $\mathbf{e}$. If so, then since A_r produces the empty word, so does A_i, so we append i to $\mathbf{e}$. If not, then we change the value of lP to i and that of oL to false. We then return to the beginning of the loop to proceed with the next i.

The second possibility is that the production of the rule is a terminal character. Recall that we added all of the production rules which produce terminal characters to $\mathbf{Q}$ in the first small loop, so in this case no rules get added to $\mathbf{Q}$. When the production of A_i is a terminal character, we first check to see if the production is the empty letter. If it is, then we increase the value of k by 1 and append i to the list $\mathbf{e}$. If the production is not the empty letter, then the production rule must be of the form $A_i \to a_r^{\pm 1}$ for some r. If $a_r^{\pm 1}$ is already in the list $\mathbf{l}$, then A_i is not the first non-terminal character encountered whose production is $a_r^{\pm 1}$, so we increase the value of k by 1. If $a_r^{\pm 1}$ does not appear in $\mathbf{l}$, then we append it to $\mathbf{l}$. In every case where the production of A_i is a terminal character, we set $\mathbf{z}[i]$ to be $i - \mathbf{z2}[i]$ because $\mathbf{z2}[i]$ is the index in $\mathbf{Q}$ of that terminal character. (In the case of the empty letter, the index in $\mathbf{Q}$ is -1, indicating that a production of the empty letter does not occur in $\mathbf{Q}$.) Then we return to the beginning of the loop.

The third and final possibility is that the rule is of the form $A_i \to A_r \cdot A_s$ for some r and s. If r and s are both in $\mathbf{e}$, then A_i produces the empty letter, so we increase the value of k by 1, append i to $\mathbf{e}$, and change $\mathbf{z}[i]$ to $i + 1$. If r occurs in $\mathbf{e}$ but s does not, we change $\mathbf{z}[i]$ to $i - s + z[s]$ since A_i produces the same word that A_s does; similarly, if s is in $\mathbf{e}$ but r is not, we change $\mathbf{z}[i]$ to $i - r + z[r]$. If either r or s is in $\mathbf{e}$ but not both, we then set oL to false, set lP to i, and increase the value of k by 1. Finally, if neither r nor s is in $\mathbf{e}$ then first we append $\mathbf{Q}[r - \mathbf{z}[r]] \cdot \mathbf{Q}[s - \mathbf{z}[s]]$ to $\mathbf{Q}$, indicating the new rule $B_q \to B_{r-\mathbf{z}[r]} \cdot B_{s-\mathbf{z}[s]}$. (Technically, we

append a Concat-type ProdRule with attributes $idx1 = r - \mathbf{z}[r]$ and $idx2 = s - \mathbf{z}[s]$ to $\mathbf{Q}$, but we use the description in the previous sentence for the sake of readability.) We also change $\mathbf{z}[i]$ to k since $i - k$ is the index in $\mathbf{Q}$ of this new rule; let $\mathbf{CL}[q] = 1$, $oL =$ false, and $lC = i$; and increase the value of q by 1. We return to the beginning of the loop.

After progressing through all of the production rules of $\mathbb{A}$, we exit the main loop of the algorithm. (At this point we return to the flowchart in Figure 2.4.) If the index $n - 1$ occurs in $\mathbf{e}$, then the root of $\mathbb{A}$, that is, A_{n-1}, produces the empty word. In this case, we set oL to true. If $n - 1$ is not in $\mathbf{e}$, and if $lP > lC$, then w_A is not empty, but there was no ProdRule created and appended to $\mathbf{Q}$ which corresponds to A_{n-1}. Note that it is not possible for lP and lC to be equal at this point, because if either is changed from its initial value the other cannot be changed to the same value, and if neither is changed from its initial value then the word produced by A_{n-1} is empty and so we do not reach this point in the algorithm. Now if $lP < lC$ then lC is the highest index in $\mathbf{Q}$, and the word produced by the non-terminal character B_{lC} corresponding to $\mathbf{Q}[lC]$ is the same as the word produced by A_{n-1}, so no further editing of $\mathbf{Q}$ is required. If $lP > lC$, we need to determine which ProdRule in $\mathbf{Q}$ produces the same word as that produced by A_{n-1}. To this end, we perform the following steps.

We set i to lP, because when $lP > lC$, lP is the highest index of a non-terminal character in $\mathbb{A}$ which produces a nonempty word. We set the variable nr (short for 'next rule') to $i - \mathbf{z}[i]$ and check to see if $\mathbf{CL}[nr] = 0$. Note that nr is the index in $\mathbf{Q}$ of the production that corresponds to that of A_i. If $\mathbf{CL}[nr] \neq 0$, this means that $\mathbf{Q}[nr]$, the production rule corresponding to that of A_i, is a Concat-type ProdRule, and thus the word produced by $\mathbf{Q}[nr]$ is the same as that produced by A_i, and by extension, A_{n-1}. If this is the case, then we truncate $\mathbf{Q}$ so that it contains only its first $nr + 1$ items, which indicate our new production rules; this causes B_{nr}, which corresponds to $\mathbf{Q}[nr]$, to be the root. If, on the other hand, $\mathbf{CL}[nr] = 0$, then $\mathbf{Q}[nr]$ is a terminal character. If this is the case, then A_i, and by extension, A_{n-1}, produces the single letter $\mathbf{Q}[nr]$. Thus we append to $\mathbf{Q}$ the Pr1Pr-type ProdRule with

attribute $idx1$ set to nr, corresponding to the rule $B_q \to B_{nr}$, where q is the length of $\mathbf{Q}$ before appending this rule.

Before exiting the routine we check to see if oL is true. If so, we set $\mathbf{Q}$ to [ ], the empty list. In either case, as described in the previous two paragraphs, the word produced by the item of highest index in $\mathbf{Q}$ is the same as w_A. Thus we create a new SLP $\mathbb{B}$ with the set $\mathbf{Q}$ of production rules and then output $\mathbb{B}$.

Now $\mathbb{B}$ meets all the criteria for being in quasinormal form: The rules whose productions are terminal characters have smaller indices than those which produce non-terminal characters, because we append those rules whose productions are terminal characters, and no other rules, to $\mathbf{Q}$ in the small loop at the beginning, before running the main loop in which those with productions containing non-terminal characters are appended to $\mathbf{Q}$. By design, as described above, the empty word does not occur in any production in $\mathbf{Q}$, and there is a Pr1Pr-type ProdRule in $\mathbf{Q}$ if and only if w_A is a single letter. □

Lemma 2.11. *The algorithm Quasinormalize SLP described by the flowcharts in Figures 2.4 and 2.5 runs in polynomial time in n, the length of the SLP which is input.*

Proof. The operations that happen before entering the first small loop and before entering the main loop, as well as the time required to determine whether or not $i < n$, do not depend on n and so happen in constant time, say c_1 steps. For each iteration of the first loop, the time is independent of n, and so the number of steps is bounded by a constant, say c_2. The first loop runs n times, so the total number of steps before running the main loop is no more than $c_1 + c_2 n$, which is linear. The main loop also runs n times, and the time required for each iteration is independent of n; say c_3 is a bound for the number of steps each iteration takes. The number of steps after running the main loop is also independent of n and therefore runs in constant time, say c_4 steps. Therefore the total number of steps required to run Quasinormalize SLP is $c_1 + c_2 n + c_3 n + c_4$, which is linear in n. Because of Remark 2.3, this

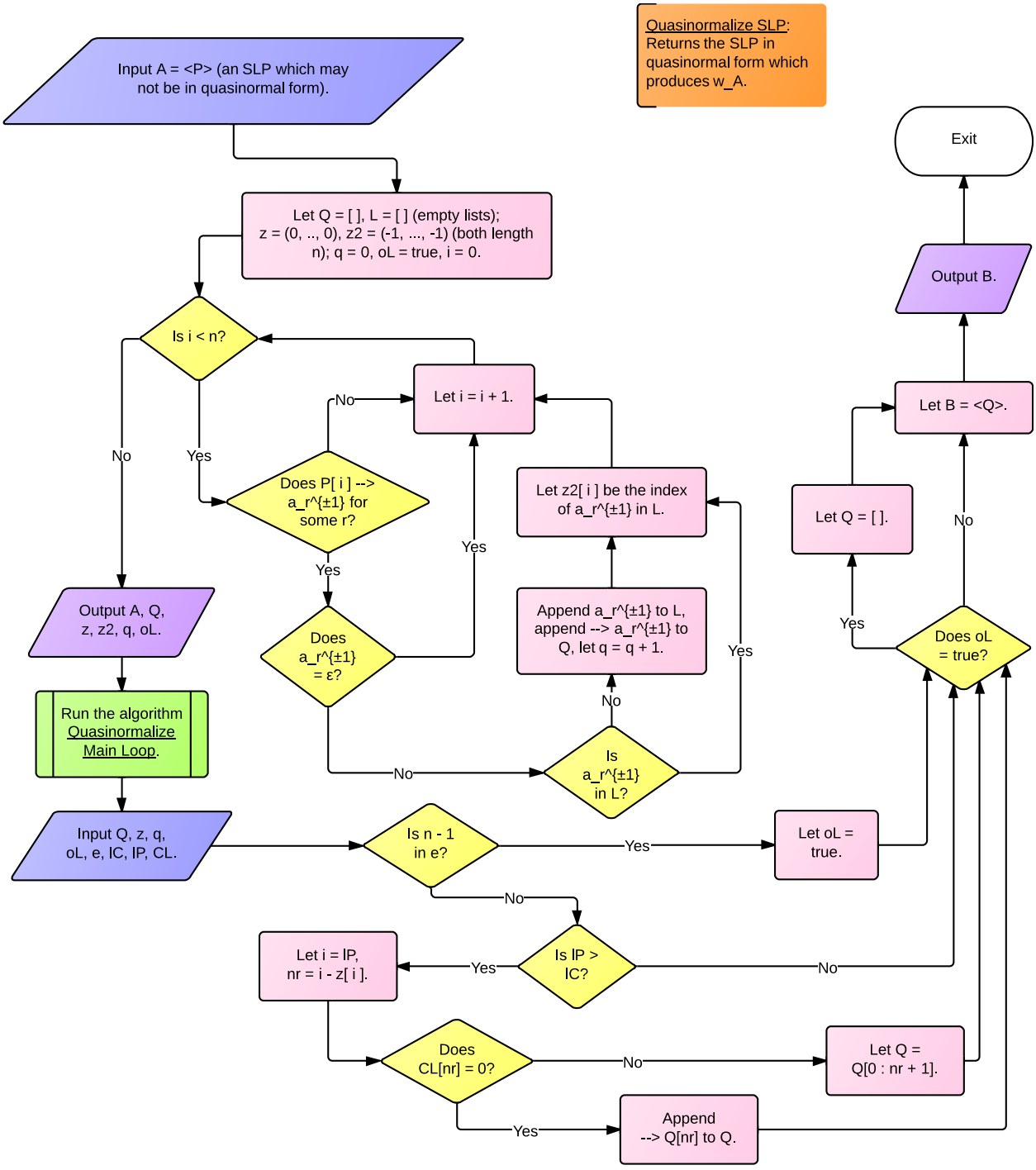


Figure 2.4: Algorithm Quasinormalize SLP

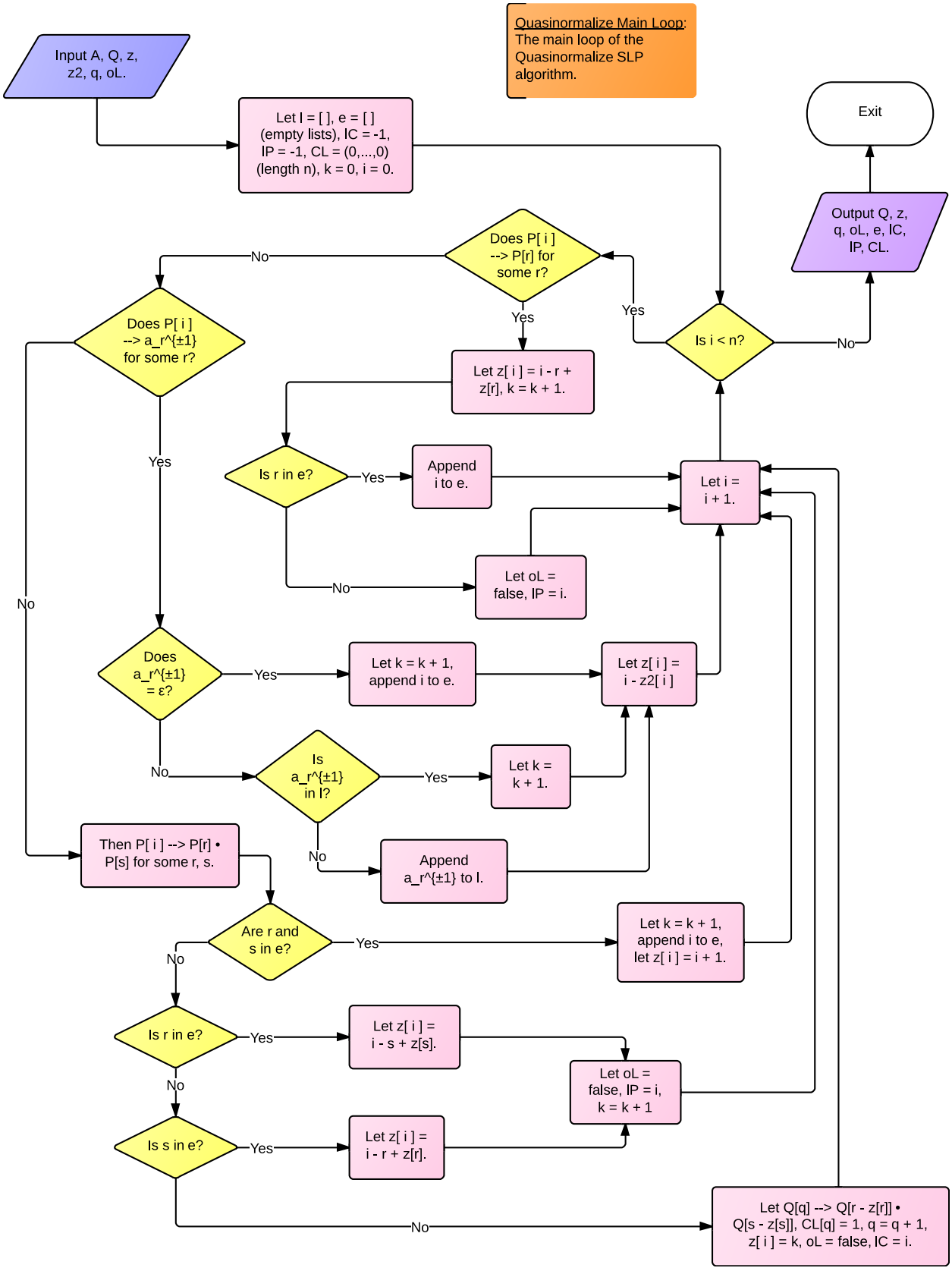


Figure 2.5: Main Loop of Algorithm Quasinormalize SLP

time is not actually linear, but on the order of $n \log n$. Therefore Quasinormalize SLP runs in polynomial time. □

Lemma 2.12. *The length of the SLP produced by Quasinormalize SLP is no more than n, where n is the length of the SLP which is input.*

Proof. By construction, for each production rule of the SLP which is input into Quasinormalize SLP, we create at most one new production rule, so the number of production rules may decrease or stay the same, but it may not increase. Thus the length of $\mathbb{B}$, the SLP we create with the new production rules, is no more than the length of the SLP which was input. □

Example 2.13. In this example we will consider how the algorithm Quasinormalize SLP described by the flowcharts in Figures 2.4 and 2.5 works when we input a particular SLP, and what output is produced in this case. For this example, let $m = 5$, and let the SLP we input be called $\mathbb{C}$ and have the following attributes: **gens** = [‘$a0$’, ‘$a1$’, ‘$a2$’, ‘$a3$’, ‘$a4$’]; **invgens** = [‘$A0$’, ‘$A1$’, ‘$A2$’, ‘$A3$’, ‘$A4$’]; **P**, which we will describe in a table below; and $numbr = 34$. We will use a table to illustrate the list **P** in order to allow the reader to easily see the index of each entry. In the ProdRule columns of the table representing **P** below, a string in single quotes indicates a ProdRule object with attribute wi equal to that string; a single number indicates a ProdRule with wi being a Pr1Pr object having that number as its $idx1$ attribute; and two numbers with a dot in between indicates a ProdRule with wi being a Concat object having the first number as its $idx1$ attribute and the second number as its $idx2$ attribute. Note that the string ‘$ee00$’ is the representation of the empty letter used in our routines.

The list **P**

Index	ProdRule	Index	ProdRule	Index	ProdRule	Index	ProdRule
0	'$a0$'	8	'$A3$'	17	'$a0$'	26	'$A4$'
1	'$a1$'	9	'$A4$'	18	'$a1$'	27	18
2	'$a2$'	10	2	19	'$a2$'	28	'$ee00$'
3	'$a3$'	11	9	20	'$a3$'	29	$27 \cdot 28$
4	'$a4$'	12	0	21	'$a4$'	30	$28 \cdot 29$
5	'$A0$'	13	10	22	'$A0$'	31	$30 \cdot 27$
6	'$A1$'	14	$11 \cdot 13$	23	'$A1$'	32	$16 \cdot 31$
7	'$A2$'	15	$12 \cdot 12$	24	'$A2$'	33	32
		16	$14 \cdot 15$	25	'$A3$'		

There are three reasons that $\mathbb{C}$ is not in quasinormal form: the first is that the strings representing the generators and their inverses all appear twice in the list of production rules, with some of the them appearing after rules whose productions contain non-terminal characters; the second is that the empty letter appears in the list of production rules; and the third is that there are several Pr1Pr-type ProdRules. (Recall that a Pr1Pr-type ProdRule is only allowed if the string produced by the SLP is a single letter.) The tree representing $\mathbb{C}$ is drawn in Figure 2.6 below, with those characters whose productions are either a single non-terminal character or the empty letter printed in red. As we proceed through the proof, we will remark upon how the algorithm acts upon this example in particular.

We first discuss what happens during the first loop in our example. The first ten production rules encountered are strings of the form 'ar' or 'Ar' for some $r \in \{0, 1, 2, 3, 4\}$, so at the end of the loop when $i = 9$, the list **L** is ['$a0$', '$a1$', '$a2$', '$a3$', '$a4$', '$A0$', '$A1$', '$A2$', '$A3$', '$A4$']. Furthermore, for $i = 0, 1, \ldots, 10$, $\mathbf{Q}[i]$ is a ProdRule whose wi attribute is the same as $\mathbf{L}[i]$. In other words, we could represent **Q** as ['$a0$', '$a1$', '$a2$', '$a3$', '$a4$', '$A0$', '$A1$', '$A2$',

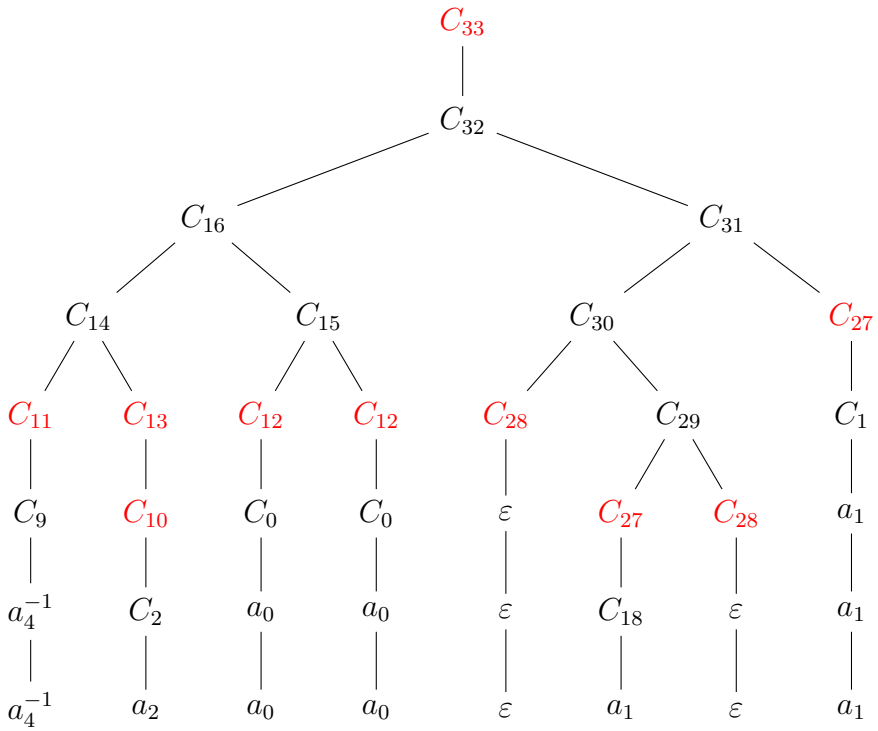


Figure 2.6: An SLP Not in Quasinormal Form

'$A3$', '$A4$'], with the understanding that we are using the same kind of representation for ProdRules as we did with **P** in the table above. Additionally, q is now equal to 10, and $\mathbf{z2} = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, -1, -1, \ldots, -1]$. The next seven production rules are ignored by our loop, because their productions contain non-terminal characters. The following ten have the same productions as the first ten, so **L**, **Q**, and q are unchanged, but **z2** becomes [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, −1, −1, −1, −1, −1, −1, −1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, −1, −1, −1, −1, −1, −1, −1, −1, −1]. For $i = 27$, the rule $C_{27} \to C_{18}$ is ignored, but for $i = 28$, the rule is $C_{28} \to$ '$ee00$'. However, this is also ignored since the production is the empty letter. The remaining 5 rules are also ignored because their productions contain non-terminal characters. So in our example, at the end of the first loop, **L** = ['$a0$', '$a1$', '$a2$', '$a3$', '$a4$', '$A0$', '$A1$', '$A2$', '$A3$', '$A4$']; **Q** = ['$a0$', '$a1$', '$a2$', '$a3$', '$a4$', '$A0$', '$A1$', '$A2$', '$A3$', '$A4$']; $q = 10$, and **z2** = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, −1, −1, −1, −1, −1, −1, −1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, −1, −1, −1, −1, −1, −1, −1, −1, −1].

Let us now consider how the main loop of the algorithm acts upon $\mathbb{C}$. (This is the part of the routine illustrated in the flowchart in Figure 2.5.) The first ten rules encountered all have non-empty terminal productions, and each is distinct, so each production is appended to $\mathbf{l}$ in turn. At the end of the first ten steps, therefore, $\mathbf{l} = [$'$a0$', '$a1$', '$a2$', '$a3$', '$a4$', '$A0$', '$A1$', '$A2$', '$A3$', '$A4$'$]$. The first ten items of $\mathbf{z}$ are set to 0, and all other variables are left unchanged by the first ten steps through the main loop.

The next four rules we meet are $C_{10} \to C_2$, $C_{11} \to C_9$, $C_{12} \to C_0$, and $C_{13} \to C_{10}$. In these steps $\mathbf{z}$ gets changed so that $\mathbf{z}[10] = 10-2+0 = 8$, $\mathbf{z}[11] = 11-9+0 = 2$, $\mathbf{z}[12] = 12-0+0 = 12$, and $\mathbf{z}[13] = 13 - 10 + 8 = 11$. The value of k is increased at each of these four steps, so that $k = 4$ at the end of them. Since the words produced by C_{10}, C_{11}, C_{12}, and C_{13} are all non-empty, oL is set to false and lP is set to the current value of i at each step, making $lP = 13$ at the end of these four steps.

Next we encounter $C_{14} \to C_{11} \cdot C_{13}$, $C_{15} \to C_{12} \cdot C_{12}$, and $C_{16} \to C_{14} \cdot C_{15}$. None of these non-terminal characters produces the empty word, so the corresponding rules are added to $\mathbf{Q}$ and the corresponding values of $\mathbf{z}$ are changed: For $C_{14} \to C_{11} \cdot C_{13}$, we have $r = 11$ and $s = 13$, so we append $\mathbf{Q}[11-2] \cdot \mathbf{Q}[13-11] = \mathbf{Q}[9] \cdot \mathbf{Q}[2]$ to $\mathbf{Q}$, indicating the new rule $D_{10} \to D_9 \cdot D_2$, and we change $\mathbf{z}[14]$ to 4 since $k = 4$ at this point. Similarly, for $C_{15} \to C_{12} \cdot C_{12}$, we append $\mathbf{Q}[12-12] \cdot \mathbf{Q}[12-12] = \mathbf{Q}[0] \cdot \mathbf{Q}[0]$ to $\mathbf{Q}$, indicating the new rule $D_{11} \to D_0 \cdot D_0$, and we change $\mathbf{z}[15]$ to 4. For $C_{16} \to C_{14} \cdot C_{15}$, we append $\mathbf{Q}[14-4] \cdot \mathbf{Q}[15-4] = \mathbf{Q}[10] \cdot \mathbf{Q}[11]$ to $\mathbf{Q}$, indicating the new rule $D_{12} \to D_{10} \cdot D_{11}$, and we change $\mathbf{z}[16]$ to 4. At each of these three steps, the value of $\mathbf{CL}[q]$ is set to 1, q is increased by 1, oL is set to false, and lC is set to the current value of i. Thus at the end of these three steps, items 10, 11, and 12 of $\mathbf{CL}$ are 1 and all other items are 0; $q = 13$, oL is false, and $lC = 16$.

For the sake of brevity we will not expound upon all seventeen remaining production rules, but rather highlight a few points. At the end of the step for $i = 26$, k has increased by 10 and so is now 14, items 17 through 26 of $\mathbf{z}$ are set to 17, and no other variables have

changed further. When $i = 27$, $\mathbf{z}[27]$ is set to 26; k becomes 15, and lP is changed to 27. When $i = 28$, the first production rule with an empty production is encountered, so $\mathbf{e}$ has its first entry, the value 28, appended; $\mathbf{z}[28]$ is set to 29; and k increases to 16. When $i = 29$, we encounter the rule $C_{29} \to C_{27} \cdot C_{28}$, but since C_{28} produces the empty word, we change $\mathbf{z}[29]$ to $29 - 27 + 26 = 28$, set lP to 29, and increase k to 17. Similarly, when $i = 30$, $\mathbf{z}[30]$ is set to $30 - 29 + 28 = 29$, lP increases to 30, and k becomes 18. At the end of the step for $i = 32$, two more entries have been added to $\mathbf{Q}$ which indicate the rules $D_{13} \to D_1 \cdot D_1$ and $D_{14} \to D_{12} \cdot D_{13}$; items 13 and 14 of $\mathbf{CL}$ have been set to 1; $q = 15$; $lC = 32$; and $\mathbf{z}[31] = \mathbf{z}[32] = 18$. The final step is for $i = 33$, during which k is increased to 19, lP is set to 33, and $\mathbf{z}[33]$ is changed to 19. Thus at the end of the main loop, $lP = 33$, $lC = 32$, $q = 15$, $oL =$ false, the list $\mathbf{e}$ has a single entry: $\mathbf{e} = [28]$, and the values of $\mathbf{Q}$, $\mathbf{z}$, and $\mathbf{CL}$ for each index are given in the table below.

The lists **Q**, **z**, and **CL**

Index	**Q**	**z**	**CL**	Index	**Q**	**z**	**CL**	Index	**Q**	**z**	**CL**
0	'$a0$'	0	0	11	$0 \cdot 0$	2	1	23		17	0
1	'$a1$'	0	0	12	$10 \cdot 11$	12	1	24		17	0
2	'$a2$'	0	0	13	$1 \cdot 1$	11	1	25		17	0
3	'$a3$'	0	0	14	$12 \cdot 13$	4	1	26		17	0
4	'$a4$'	0	0	15		4	0	27		26	0
5	'$A0$'	0	0	16		4	0	28		29	0
6	'$A1$'	0	0	17		17	0	29		28	0
7	'$A2$'	0	0	18		17	0	30		29	0
8	'$A3$'	0	0	19		17	0	31		18	0
9	'$A4$'	0	0	20		17	0	32		18	0
10	$9 \cdot 2$	8	1	21		17	0	33		19	0
				22		17	0				

Now $34 - 1 = 33$ does not occur in **e**, and $lP = 33 > lC = 32$. So we let $i = 33$ and $nr = 33 - 19 = 14$. Since $\mathbf{CL}[14] = 1$, we truncate **Q** to 15 items so that $D_1 4$, the character associated with $\mathbf{Q}[14]$, is the root. Since oL is false, we create a new SLP $\mathbb{D}$ with the set **Q** of production rules and then output $\mathbb{D}$. The tree representing $\mathbb{D}$ is pictured in Figure 2.7.

We now review several simpler algorithms upon which our main algorithms are built.

Lemma 2.14. *The algorithm Inverse SLP described by the flowchart in Figure 2.8 returns an SLP in quasinormal form which produces the inverse of the word produced by the SLP which is input.*

Proof. We input an SLP $\mathbb{A}$ with production rule list **P**, and we create a list, **Q**, of length n, setting each of its items to ε. We run the algorithm Get Length on $\mathbb{A}$ to find the length of w_A

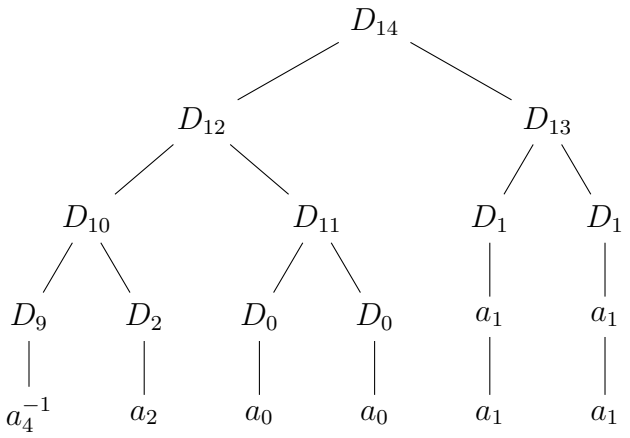


Figure 2.7: The Output of Quasinormalize in Example 2.13

by Lemma 2.4. We use this to determine whether or not any production rule points to only one non-terminal character; since $\mathbb{A}$ is in quasinormal form, there is a Pr1Pr-type production rule in $\mathbf{P}$ if and only if w_A is a single letter. Each item in $\mathbf{Q}$ corresponds to a non-terminal character, and we use the index q to step through the production rules of $\mathbb{A}$ one at a time, starting with $q = 0$. Since $\mathbb{A}$ is in quasinormal form, the first production rules encountered are those pointing to terminal characters. For these, if the rule is $A_q \to a_r$ for some r, then $\mathbf{Q}[q]$ is set to a_r^{-1}, which indicates the new production rule $B_q \to a_r^{-1}$; if the rule is $A_q \to a_r^{-1}$ for some r, then $\mathbf{Q}[q]$ is set to a_r, indicating the new rule $B_q \to a_r$. This has the effect of producing the inverse of each letter in w_A. If the production rule is of the type $A_q \to A_r$, then $\mathbf{Q}[q]$ is set to $\mathbf{Q}[r]$, indicating the new rule $B_q \to B_r$. All other production rules must be of the form $A_q \to A_r \cdot A_s$. In this case, $\mathbf{Q}[q]$ is set to $\mathbf{Q}[s] \cdot \mathbf{Q}[r]$, indicating the new rule $B_q \to B_s \cdot B_r$. This has the effect of producing letters in the reverse order of the letters in w_A. Now for every production rule of $\mathbb{A}$ pointing to a letter we have a new production rule pointing to the inverse of that letter, and for every production rule pointing to two non-terminal characters we have a new production rule pointing to the same characters but in reverse order. We create a new SLP with this new set of production rules and run the algorithm Quasinormalize SLP on it, so the new SLP is in quasinormal form by Lemma 2.10, and the word produced by this new SLP is the inverse of the word produced by $\mathbb{A}$. □

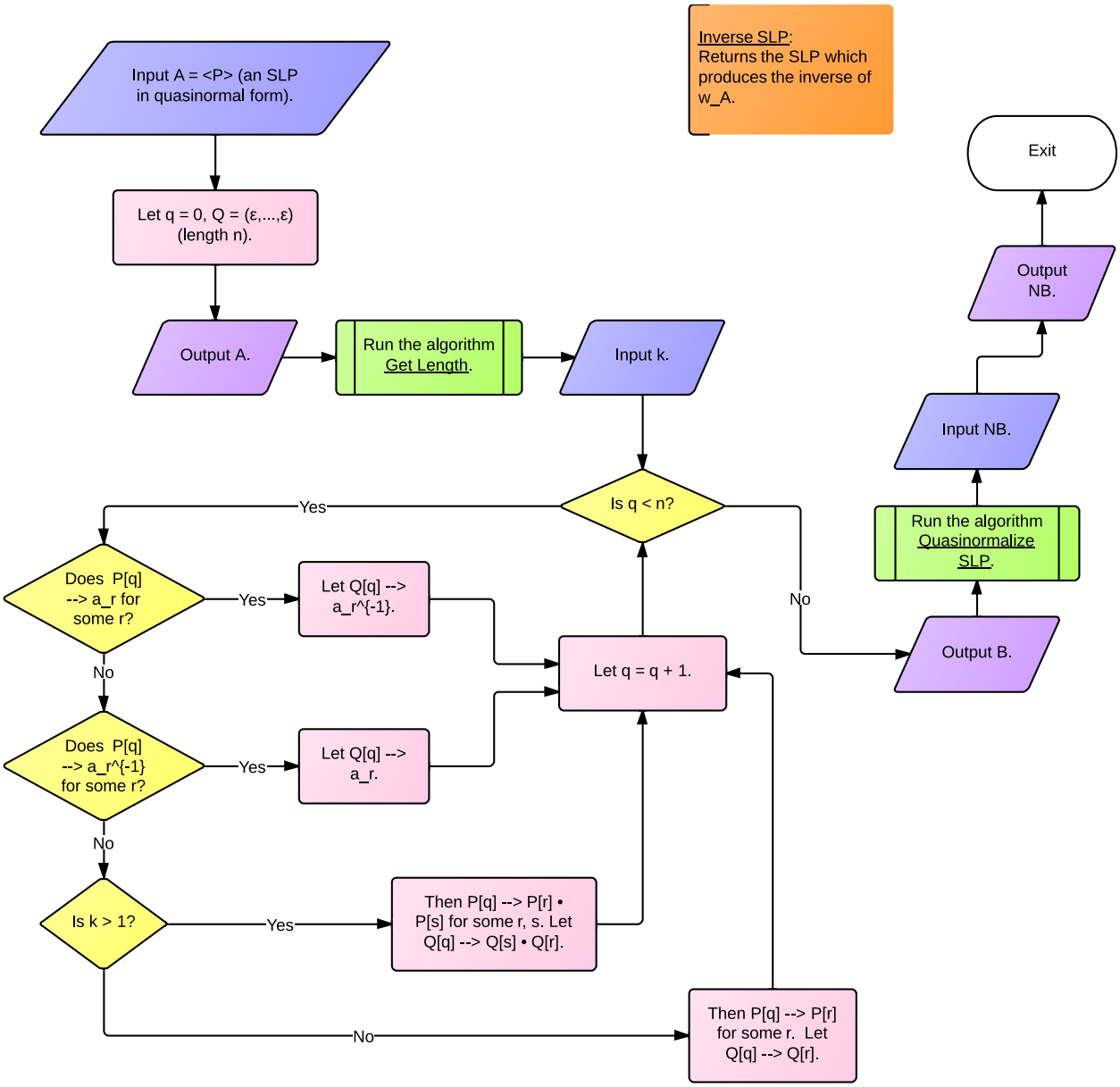


Figure 2.8: Algorithm Inverse SLP

Lemma 2.15. *The algorithm Inverse SLP described by the flowchart in Figure 2.8 runs in polynomial time in n, the length of the SLP which is input.*

Proof. The time it takes to run the algorithm Get Length is polynomial, say $Q(n)$, by Lemma 2.5. The other operations that happen outside the main loop before quasinormalizing, as well as the time required to determine whether or not $q < n$, do not depend on n and so happen in constant time, say c_1 steps. For each iteration of the main loop, the time is independent of n, and so the number of steps is bounded by a constant, say c_2. The main loop runs n times, so the total number of steps before quasinormalizing is no more than $Q(n) + c_1 + c_2 n$, which is polynomial. By construction, the new SLP which is created has length n, the same length as A. The time it takes to run the algorithm Quasinormalize SLP is polynomial, say $p(n)$, by Lemma 2.11. Adding $Q(n)$ to the polynomial $p(n)$ results in a new polynomial in n. This is the time required to run Inverse SLP. □

Lemma 2.16. *The algorithm Reverse SLP described by the flowchart in Figure 2.9 inputs an SLP $\mathbb{A}$ and returns an SLP in quasinormal form which produces w_A in reverse.*

Proof. This routine is very similar to the Inverse SLP algorithm, so we will merely recall the main points of Inverse SLP and highlight the differences between it and Reverse SLP and how they affect the outcome of the routine. As with Inverse SLP, we input an SLP $\mathbb{A}$ with production rule list $\mathbf{P}$, and we create a list, $\mathbf{Q}$, of length n, setting each of its items to ε. We do not need to find the length of w_A as we did in Inverse SLP because this piece of Put In Lexicographic Order is only encountered if w_A has more than one letter; therefore we know there are no Pr1Pr-type production rules in $\mathbf{P}$. Each item in $\mathbf{Q}$ corresponds to a non-terminal character, and we use the index q to step through the production rules of $\mathbb{A}$ one at a time, starting with $q = 0$. Since $\mathbb{A}$ is in quasinormal form, the first production rules encountered are those pointing to terminal characters. For these, unlike in Inverse SLP, we merely copy them to $\mathbf{Q}$, so unlike the output of Inverse SLP, the output contains the same letters as

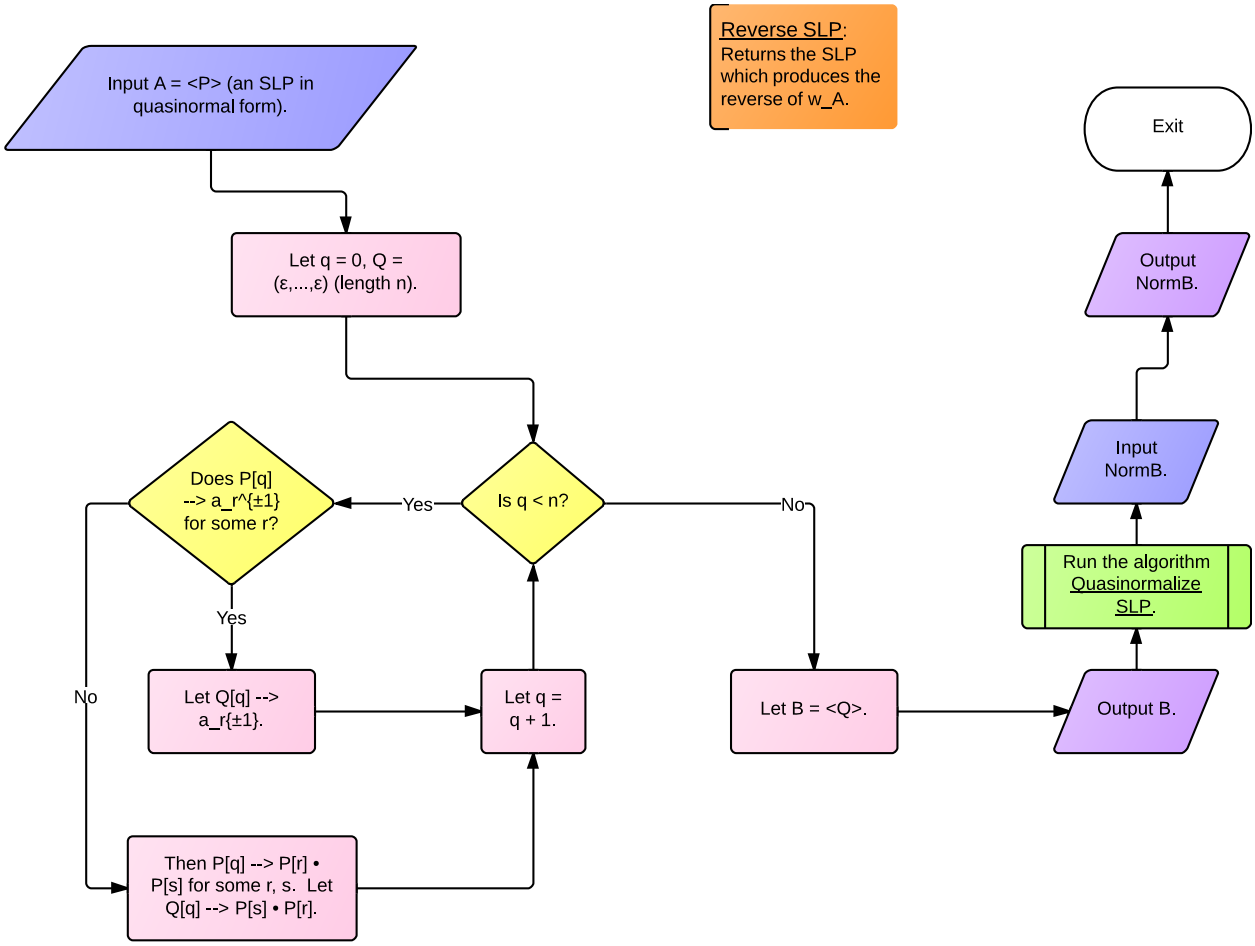


Figure 2.9: Algorithm Reverse SLP

w_A rather than the inverse of each letter. All other production rules must be of the form $A_q \to A_r \cdot A_s$. In this case, as in Inverse SLP, $\mathbf{Q}[q]$ is set to $\mathbf{Q}[s] \cdot \mathbf{Q}[r]$, indicating the new rule $B_q \to B_s \cdot B_r$. This has the effect of producing letters in the reverse order of the letters in w_A. Now for every production rule of $\mathbb{A}$ pointing to a letter we have a new production rule pointing to that letter, and for every production rule pointing to two non-terminal characters we have a new production rule pointing to the same characters but in reverse order. We create a new SLP with this new set of production rules and run the algorithm Quasinormalize SLP on it, so the new SLP is in quasinormal form by Lemma 2.10, and the word produced by this new SLP is the word produced by $\mathbb{A}$ in reverse order. □

Lemma 2.17. *The algorithm Reverse SLP described by the flowchart in Figure 2.9 runs in*

polynomial time in n, the length of the SLP which is input.

Proof. All of the steps in Reverse SLP are also in Inverse SLP (but not all of the steps in Inverse SLP are in Reverse SLP). Since Inverse SLP runs in polynomial time by Lemma 2.15, so does Reverse SLP. □

Lemma 2.18. *The algorithm Count the Occurrences described by the flowchart in Figure 2.10 inputs an SLP $\mathbb{A}$ and a nonnegative integer $u < m$ and returns the total number of occurrences of the generator a_u and its inverse a_u^{-1} in w_A.*

Proof. We input an SLP $\mathbb{A}$, and a nonnegative integer $u < m$. We create a list, $\mathbf{Q}$, of length n, setting each list item to ε. Each item in $\mathbf{Q}$ corresponds to a non-terminal character, and we use the index q to step through the production rules of $\mathbb{A}$ one at a time, starting with $q = 0$. Since $\mathbb{A}$ is in quasinormal form, the first production rules encountered are those pointing to terminal characters. For these, if the rule is $A_q \to a_u$ or $A_q \to a_u^{-1}$, then $\mathbf{Q}[q]$ is set to a_u or a_u^{-1}, respectively, which indicates the new production rule $B_q \to a_u$ or $B_q \to a_u^{-1}$. If the rule is $A_q \to a_r^{\pm 1}$ for some $r \neq u$, then $\mathbf{Q}[q]$ is set to the empty letter ε, indicating the new rule $B_q \to \varepsilon$. This has the effect of removing every letter in w_A from our new word except $a_u^{\pm 1}$. All other production rules are merely copied: If the production rule is of the type $A_q \to A_r$, then $\mathbf{Q}[q]$ is set to $\mathbf{Q}[r]$, indicating the new rule $B_q \to B_r$, and if the production rule is of the form $A_q \to A_r \cdot A_s$, then $\mathbf{Q}[q]$ is set to $\mathbf{Q}[r] \cdot \mathbf{Q}[s]$, indicating the new rule $B_q \to B_r \cdot B_s$. We create a new SLP $\mathbb{B}$ with this new set of production rules and run the algorithm Quasinormalize SLP on it. By Lemma 2.10, the new SLP returned, $\mathbb{NB}$, produces the same word as $\mathbb{B}$ and is in quasinormal form. We then run Get Length on $\mathbb{NB}$ to find, by Lemma 2.4, the length k of the word it produces. Since the only letters in this new word are a_u and a_u^{-1}, and since we have exactly the same number of occurrences of each of these in the new word as there are in w_A, k is the total number of occurrences of a_u and a_u^{-1} in w_A. We output k. □

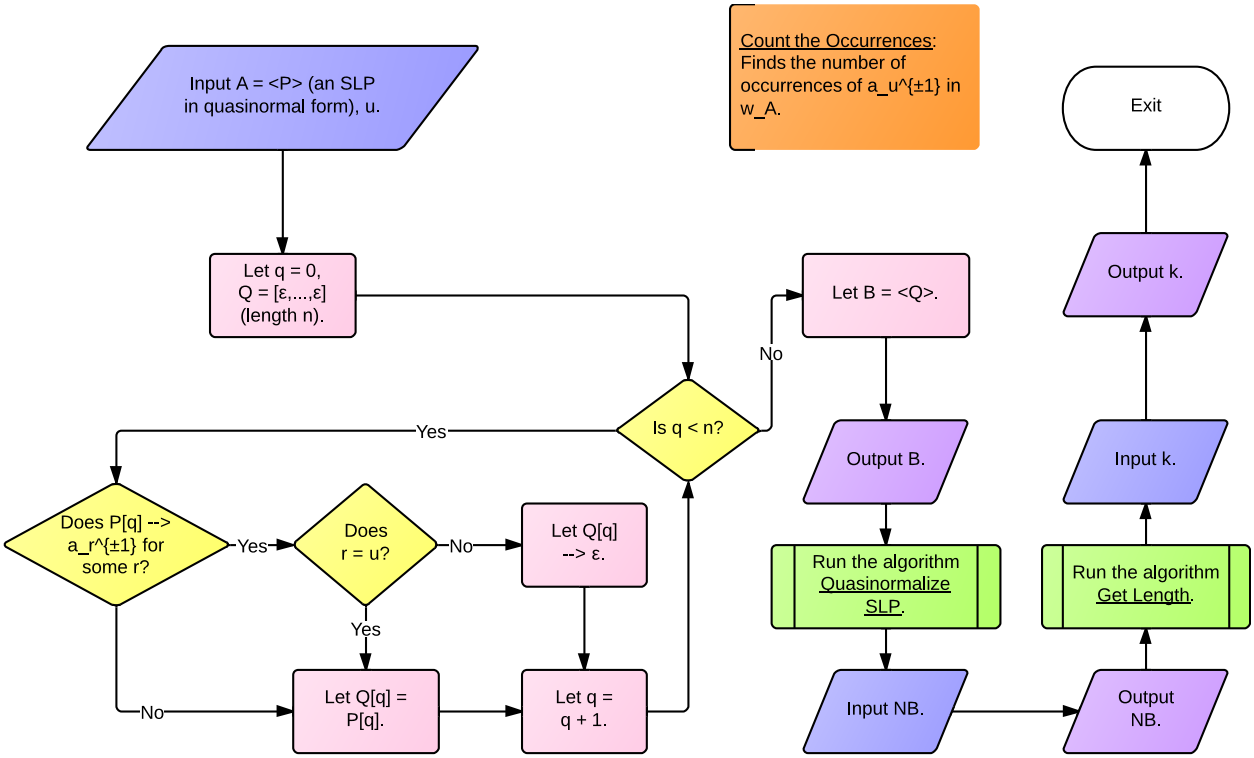


Figure 2.10: Algorithm Count the Occurrences

Lemma 2.19. *The algorithm Count the Occurrences described by the flowchart in Figure 2.10 runs in polynomial time in n, the length of the SLP which is input.*

Proof. The time it takes to run the operations that happen outside the main loop before running Quasinormalize SLP, as well as the time required to determine whether or not $q < n$, do not depend on n and so happen in constant time, say c_1 steps. For each iteration of the main loop, the time is independent of n, and so the number of steps is bounded by a constant, say c_2. The main loop runs n times, so the number of steps before quasinormalizing is no more than $c_2 n + c_1$. By construction, the new SLP, $\mathbb{B}$, which is output to Quasinormalize SLP, has length n. By Lemma 2.11, Quasinormalize SLP runs in polynomial time in n, say $p(n)$, and by construction returns an SLP $\mathbb{NB}$ of length no more than n. By Lemma 2.5, Get Length runs in polynomial time in n, say $q(n)$. So adding $q(n)$ to the polynomial $p(n) + c_3 n + c_4$ gives us another polynomial in n. This is the time required to run Count the

Occurrences. □

Lemma 2.20. *The algorithm Combine SLPs described by the flowchart in Figure 2.11 inputs two SLPs, $\mathbb{A}$ and $\mathbb{B}$, and returns an SLP in quasinormal form which produces the concatenation of the words produced by $\mathbb{A}$ and $\mathbb{B}$ (in that order).*

Proof. We input two SLPs, $\mathbb{A}$ and $\mathbb{B}$. We run the algorithm Get Length on $\mathbb{A}$, and if the length of w_A is 0, we output $\mathbb{B}$ and exit the routine. Otherwise, similarly, we apply Get Length to $\mathbb{B}$, and if $|w_B| = 0$, we output $\mathbb{A}$ and exit the routine. If both w_A and w_B have positive length, we continue with the rest of the program. We create a list, $\mathbf{Q}$, of length $n + p + 1$, setting each list item to ε. Each item in $\mathbf{Q}$ corresponds to a non-terminal character, and we use the index q to first step through the production rules of $\mathbb{A}$ one at a time, starting with $q = 0$. For these, we merely copy each rule to $\mathbf{Q}$: If the rule is of the form $A_q \to a_r^{\pm 1}$, then $\mathbf{Q}[q]$ is set to $a_r^{\pm 1}$, indicating the new production rule $C_1 \to a_r^{\pm 1}$. If the rule is of type $A_q \to A_r$, we set $\mathbf{Q}[q]$ to $\mathbf{Q}[r]$, indicating the new rule $C_q \to C_r$. And if the rule is $A_q \to A_r \cdot A_s$, we set $\mathbf{Q}[q]$ to $\mathbf{Q}[r] \cdot \mathbf{Q}[s]$, which indicates the new production rule $C_q \to C_r \cdot C_s$.

We next reset q to 0 and step through the production rules of $\mathbb{B}$ one at a time, stacking these on top of the rules we already have indicated in $\mathbf{Q}$. If the rule is $B_q \to a_r^{\pm 1}$ for some r, then $\mathbf{Q}[n + q]$ is set to $a_r^{\pm 1}$, indicating the new production rule $C_{n+q} \to a_r^{\pm 1}$. If the production rule is of the type $B_q \to B_r$, then $\mathbf{Q}[n+q]$ is set to $\mathbf{Q}[n+r]$, indicating the new rule $C_{n+q} \to C_{n+r}$, and if the production rule is of the form $B_q \to B_r \cdot B_s$, then $\mathbf{Q}[n+q]$ is set to $\mathbf{Q}[n+r] \cdot \mathbf{Q}[n+s]$, indicating the new rule $C_{n+q} \to C_{n+r} \cdot C_{n+s}$.

Once we have added all of the production rules from $\mathbb{A}$ and $\mathbb{B}$, we add one final item to $\mathbf{Q}$ indicating the final production rule: $\mathbf{Q}[n + p]$ is set to $\mathbf{Q}[n - 1] \cdot \mathbf{Q}[n + p - 1]$, which indicates $C_{n+p} \to C_{n-1} \cdot C_{n+p-1}$. This last rule effects the concatenation of the word produced by A_{n-1} and the word produced by B_{p-1}; that is, of w_A and w_B. We create a new SLP $\mathbb{C}$ with this new set of production rules and run the algorithm Quasinormalize SLP on it, then output

that SLP, $\mathbb{NC}$, now in quasinormal form by Lemma 2.10. Since we have merely copied the production rules of $\mathbb{A}$ and $\mathbb{B}$ and added a final production rule pointing to the concatenation of the roots of $\mathbb{A}$ and $\mathbb{B}$, $\mathbb{NC}$ will produce $w(A_{n-1}) \cdot w(B_{p-1}) = w_A \cdot w_B$. □

Lemma 2.21. *The algorithm Combine SLPs described by the flowchart in Figure 2.11 runs in polynomial time in $n + p$, where n and p are the lengths of the two SLPs which are input. Furthermore, the length of the SLP which is returned is no more than $n + p + 1$.*

Proof. By 2.5, running Get Length to find the lengths of w_A and w_B takes polynomial time in n for w_A, say $\overline{q_1}(n)$ steps, and in p for w_B, say $\overline{q_2}(p)$ steps. Thus running Get Length twice at the beginning of the routine takes no more than $\overline{q_1} + \overline{q_2}(n + p)$ steps. The time it takes to run the other operations that happen outside the two loops before running Quasinormalize SLP, as well as the time required to determine whether or not $q < n$ or $q < p$, do not depend on n or p and so happen in constant time, say c_1 steps. For each iteration of the first loop, the time is independent of n and p, and so the number of steps is bounded by a constant, say c_2. Similarly, for each iteration of the second loop, the time is independent of n and p, and so the number of steps is bounded by a constant, say c_3. Let $c = max\{c_2, c_3\}$. The first loop runs n times, and the second loop runs p times, so the number of steps before running the algorithm Quasinormalize SLP is no more than $(\overline{q_1} + \overline{q_2})(n + p) + cn + cp + c_1$, which is polynomial in $n + p$. The SLP $\mathbb{C}$ which is output to Quasinormalize SLP has length $n + p + 1$ by construction. By Lemma 2.11, the number of steps required to quasinormalize the new SLP is a polynomial in $n + p + 1$, which is also a polynomial in $n + p$. Adding $(\overline{q_1} + \overline{q_2})(n + p) + cn + cp + c_1$ to this polynomial results in a new polynomial in $n + p$. This is the time required to run Combine SLPs. Furthermore, by Lemma 2.12, the length of the SLP which is output by Combine SLPs is no more than $n + p + 1$. □

The next two routines we discuss involve finding a particular subword of the word produced by the SLP which is input. The first finds the leftmost subword of a given length and the

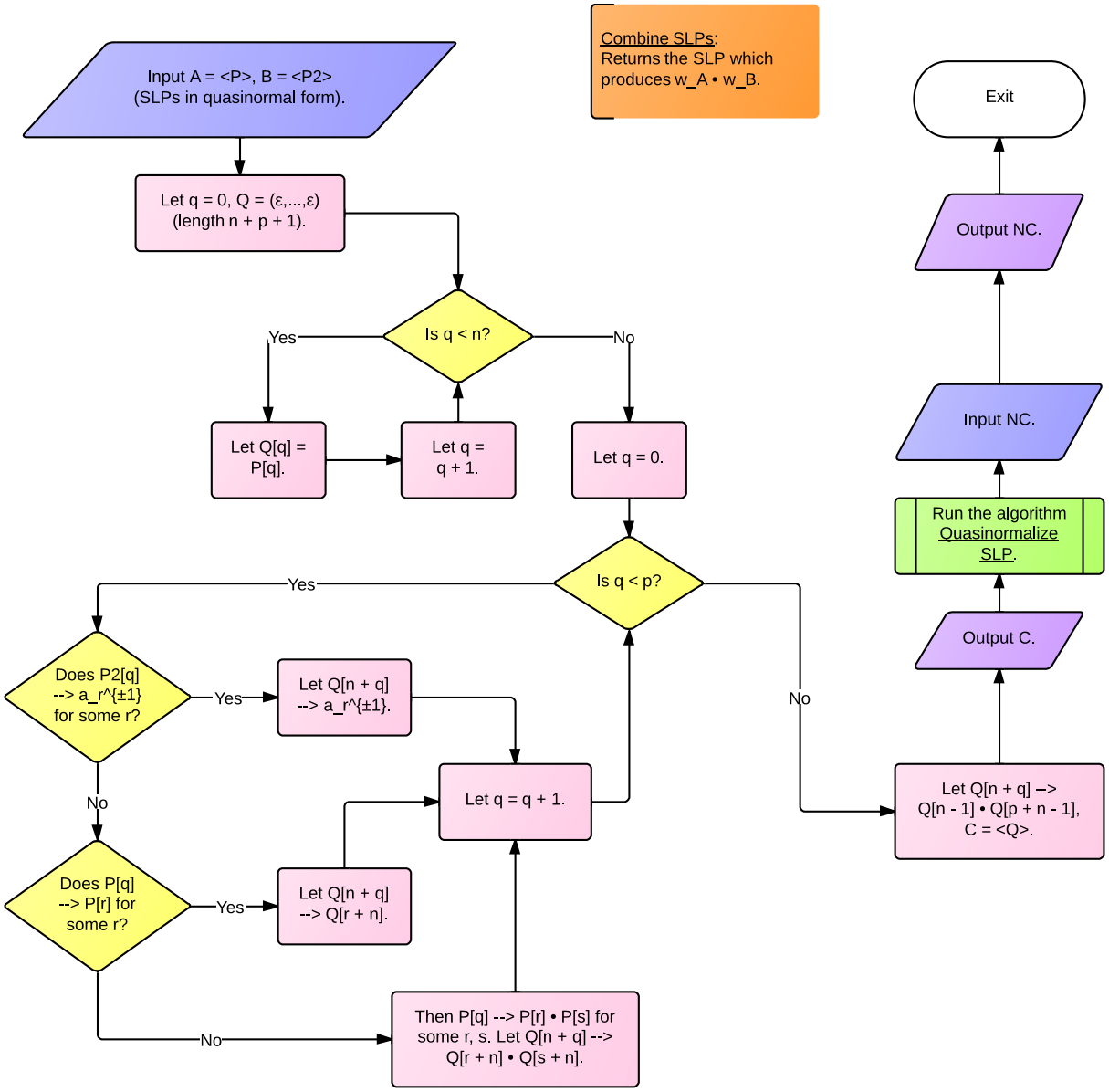


Figure 2.11: Algorithm Combine SLPs

second, which is very similar to the first, finds the rightmost subword of a given length.

Lemma 2.22. *The algorithm Left Sub SLP described by the flowchart in Figure 2.12 inputs an SLP $\mathbb{A}$ and a nonnegative integer g, and returns an SLP in quasinormal form which produces $w_A[:g]$, the leftmost subword of length g.*

Proof. Note that it may be helpful to refer to Figure 2.13 while considering this proof. In the example in this figure, $n-1=13$ and $g=12$. At the beginning of the algorithm Left Sub SLP, we input an SLP $\mathbb{A}$, and a nonnegative integer g. We run the algorithm GetLength to find the length L of w_A, and we check to see if $g > L$. If so, an error message is printed to the screen and $\mathbb{A}$ is returned. We check to see if $g = L$, and if so, $\mathbb{A}$ is returned. We also check to see if $g = 0$, and if so, an empty SLP is created and returned. If $L > g > 0$, we proceed with the rest of the routine.

We create two lists: $\mathbf{Q}$, of length $2n-1$, with all list items set to ε; and $\mathbf{prL}$, of length n, with all list items set to 0. Each item in the two lists corresponds to a non-terminal character. We use $\mathbf{Q}$ to store the new productions, and we use $\mathbf{prL}$ to store the lengths of the words produced by each original production rule. We begin by stepping through the production rules of $\mathbb{A}$ one at a time using the index q, starting with $q=0$. In this loop we double the indices of the characters, while at the same time storing lengths of words. (We will change the items of $\mathbf{Q}$ with odd index later in the routine, as needed.) Consider the following three cases. If the current production rule of $\mathbb{A}$ is $A_q \to a_r^{\pm 1}$ for some r, we set $\mathbf{prL}[q]$ to 1, and we set $\mathbf{Q}[2q]$ to $a_r^{\pm 1}$. In the second case, the rule is of the form $A_q \to A_r$, and we set $\mathbf{prL}[q]$ to $\mathbf{prL}[r]$ and $\mathbf{Q}[2q]$ to $\mathbf{Q}[2r]$. In the third case, the rule is $A_q \to A_r \cdot A_s$ for some r and s, and we set $\mathbf{prL}[q]$ to $\mathbf{prL}[r] + \mathbf{prL}[s]$ and $\mathbf{Q}[2q]$ to $\mathbf{Q}[2r] \cdot \mathbf{Q}[2s]$. Thus $\mathbf{prL}[q]$ is the length of the word produced by A_q for any q, and all of the original production rules (the ones involving A_q for some q) have a corresponding new production rule indicated in $\mathbf{Q}$, where the indices are all doubled. The items of odd index in $\mathbf{Q}$ are still set to ε.

When we are finished with that first loop, we initialize three new variables. We set the variable h to g and will use h to store the number of letters we still need to include. We create a new list, $\mathbf{u}$, of length $2n-1$, with each list item set to 0. We will use $\mathbf{u}$ to keep track of which production rules indicated in $\mathbf{Q}$ will actually get used in our new SLP. We also set the variable tf to true. We will set tf to false at the right time during the main loop to allow us to break out of that loop. Now we step through those production rules which may need to be changed in order to cut the produced word off at g letters. Beginning with $q = n-1$, we consider the corresponding production rule of $\mathbb{A}$. As we cycle through this main loop, q becomes smaller and smaller. Once we reach the first production rule which is not of the form $A_q \to A_r \cdot A_s$, we break out of the main loop and proceed to the next step. So all of the production rules considered in the main loop are of the form $A_q \to A_r \cdot A_s$. For each of these encountered, we first set the variable l to $\mathbf{prL}[r]$, the length of the word produced by A_r, and then determine whether $l > h$, $l = h$, or $l < h$.

If $l > h$, this means that $w(A_r)$ is longer than the subword we want, so we need a new production rule producing a shorter word than the one produced by A_r. We create this new production rule by setting $\mathbf{Q}[2q-1]$ to $\mathbf{Q}[2r-1]$, indicating the new production rule $B_{2q-1} \to B_{2r-1}$. We also set $\mathbf{u}[2q-1]$ and $\mathbf{u}[2r-1]$ to 1, indicating that these production rules will be used in the new SLP. We then set q to r, because we will next need to consider the production rule for A_r, and we return to the beginning of the loop.

If $l = h$, this means that $w(A_r)$ is exactly what we need to make our final subword g letters long, so we need this production rule exactly as it is. Now $\mathbf{Q}[2r]$ is already set to the production corresponding to that of A_r, so we set $\mathbf{Q}[2q-1]$ to $\mathbf{Q}[2r]$, indicating the new rule $B_{2q-1} \to B_{2r}$. We also set $\mathbf{u}[2q-1]$ and $\mathbf{u}[2r]$ to 1 and tf to false, breaking us out of the main loop to proceed to the next step.

If $l < h$, this means that $w(A_r)$ is shorter than the subword we want, so we need to keep the production rule for A_r as is and replace the production rule for A_s with a new one. Thus we

set $\mathbf{Q}[2q-1]$ to $\mathbf{Q}[2r] \cdot \mathbf{Q}[2s-1]$, indicating the new rule $B_{2q-1} \to B_{2r} \cdot B_{2s-1}$. We change $\mathbf{u}[2q-1]$, $\mathbf{u}[2r]$, and $\mathbf{u}[2s-1]$ to 1, indicating their use in the new SLP. We also set q to s, because the next rule to consider is the one for A_s. Finally, we set h to $h-l$ because $w(B_{2r})$ has length l, so we only need $h-l$ letters from $w(A_s)$, and we return to the beginning of the loop.

Perhaps a word needs to be said about why the loop terminates, since this loop is not controlled by a counter to a given integer. There are two ways the loop can terminate – either a production rule is encountered which is not of the form $A_q \to A_r \cdot A_s$, or a production rule is encountered which produces a word of length l which is equal to the variable h at that time. For the other two possibilities – that for the encountered rule $l < h$, and that for the encountered rule $l > h$ – the value of q is changed, providing a new production rule for the next run through the loop. If $l > h$, then q is set to r, which must be less than the current value of q, because $\mathbb{A}$ is in quasinormal form; similarly, if $l < h$, then q is set to s, which is less than q. So each time the loop repeats, the value of q is less than the previous time. Recall that in quasinormal form, the production rules for terminal characters have smaller indices than those for non-terminal characters, so at some point q will be small enough that the associated production rule will be for a terminal character, causing the loop to terminate, if the loop has not terminated already.

Once we exit the main loop, we set a new variable, hu, to 0 and proceed to run through one more little loop, using q as our index and letting it run from 0 to $n-1$. We want hu to be the highest index for which $\mathbf{u}$ is set to 1. To accomplish this, for each iteration of the loop we check to see if $\mathbf{u}[q] = 1$; if so, we set hu to q. Since q increases at each step of this loop, after finishing the last iteration of the loop, hu is the highest index of all items of $\mathbf{u}$ set to 1. Finally, we truncate $\mathbf{Q}$ so that it contains only its first $hu+1$ items, which indicate our new production rules; this causes B_{hu}, which corresponds to $\mathbf{Q}[hu]$, to be the root.

Consider the production rules created by this process and how they are related to the

production rules of $\mathbb{A}$. We begin by considering the rule corresponding to the root non-terminal, A_{n-1}, so at the outset $q = n-1$. Assume $A_{n-1} \to A_r \cdot A_s$ for some r and s. We have three possibilities.

If the subword we want is exactly the word produced by A_r, we set $\mathbf{Q}[2q-1]$ to $\mathbf{Q}[2r]$, indicating the new rule $B_{2q-1} \to B_{2r}$. This new rule is then used instead of the rule $B_{2q} \to B_{2r} \cdot B_{2s}$, which is our copy of $A_q \to A_r \cdot A_s$, since we mark the rule indicated in $\mathbf{Q}[2q-1]$ as used, but not the one indicated in $\mathbf{Q}[2q]$. No other rules are changed, so now $w(B_{2q-1}) = w(A_r)$. In this case we exit the main loop.

Now the second possibility: If the subword we want is a proper subword of $w(A_r)$, we set $\mathbf{Q}[2q-1]$ to $\mathbf{Q}[2r-1]$ and set q to r so that that the next time through the loop, $\mathbf{Q}[2r-1]$ will be set to the appropriate value. By doing this and marking the rule indicated in $\mathbf{Q}[2q-1]$ as used but leaving $\mathbf{Q}[2q]$ unmarked, the rule $B_{2q-1} \to B_{2r-1}$ will be included in our new SLP instead of $B_{2q} \to B_{2r} \cdot B_{2s}$.

The final case is that the subword we want contains $w(A_r)$ and a subword of $w(A_s)$. In this case, we do not change the rule for B_{2r}, because we want all of $w(B_{2r})$ in our produced word. Rather, we set $\mathbf{Q}[2q-1]$ to $\mathbf{Q}[2r] \cdot \mathbf{Q}[2s-1]$ and q to s so that the next time through the loop, $\mathbf{Q}[2s-1]$ will be set to the appropriate value. We also set h to the difference between h and the length of $w(A_r)$ so that now h indicates how many letters need to be produced by the production $\mathbf{Q}[2s-1]$. By doing this and marking the rule indicated in $\mathbf{Q}[2q-1]$ as used but not $\mathbf{Q}[2q]$, we cause the rule $B_{2q} \to B_{2r} \cdot B_{2s}$ to not be used, but the rule $B_{2q-1} \to B_{2r} \cdot B_{2s-1}$ to be used instead.

Once we have done this for A_{n-1}, we repeat the process as needed on the next production rule which must be considered. We should note here that when the process outlined above is applied to non-terminals other than the root, it is certainly possible that one occurrence (that is, the occurrence currently being considered) of A_q will correspond to B_{2q-1} in our

new rules but that another occurrence or occurrences of A_q will correspond to B_{2q}. However, only one occurrence of A_q will correspond to B_{2q-1}, because the characters indexed by odd numbers only occur on one branch of the tree, and there cannot be more than one occurrence of any given character along one branch (since the indices strictly decrease as we move along a branch from the root toward the leaves). It is fairly straightforward to see that if the main loop is exited because the remaining part of the subword which we need is produced by A_r at some step, then the word produced by B_{2q-1} is the subword of $w(A_q)$ which we wanted. On the other hand, if the loop is exited because a rule for a terminal character is encountered, then we must only have needed one more letter, which is the letter produced by that rule.

We create a new SLP $\mathbb{B}$ with this new set of rules and run the algorithm Quasinormalize SLP on it, then output that SLP, $\mathbb{NB}$, now in quasinormal form by Lemma 2.10. $\mathbb{NB}$ will now produce $w_A[:g]$. □

Lemma 2.23. *The algorithm Left Sub SLP described by the flowchart in Figure 2.12 runs in polynomial time in n, the length of the SLP, $\mathbb{A}$, which is initially input.*

Proof. The time it takes to run the algorithm Get Length is polynomial in n by Lemma 2.5, and checking to see if $g > L$, $g = L$, or $g = 0$ happens in constant time. The time required to create an empty SLP is also bounded by a constant, so if $g \geq L$ or $g = 0$, the algorithm Left Sub SLP runs in polynomial time, say $q(n)$. The other operations that happen outside the two loops before running Quasinormalize SLP, as well as the time required to determine whether or not $q < n$ or $tf = true$, do not depend on n and so happen in constant time, say c_1 steps. For each iteration of each of the three loops, the time is independent of n, so the number of steps is bounded by a constant. Take the maximum of these three constants and call it c_2. The first and third loops run n times each, and the second loop runs no more than $2n - 1$ times, so before running Quasinormalize SLP, the algorithm Left Sub SLP requires no more than $q(n) + c_1 + c_2n + c_2(2n - 1) + c_2n = q(n) + 4c_2n + c_1 - c_2$ steps. So before quasinormalizing the new SLP, $\mathbb{B}$, Left Sub SLP runs in polynomial time in n. By

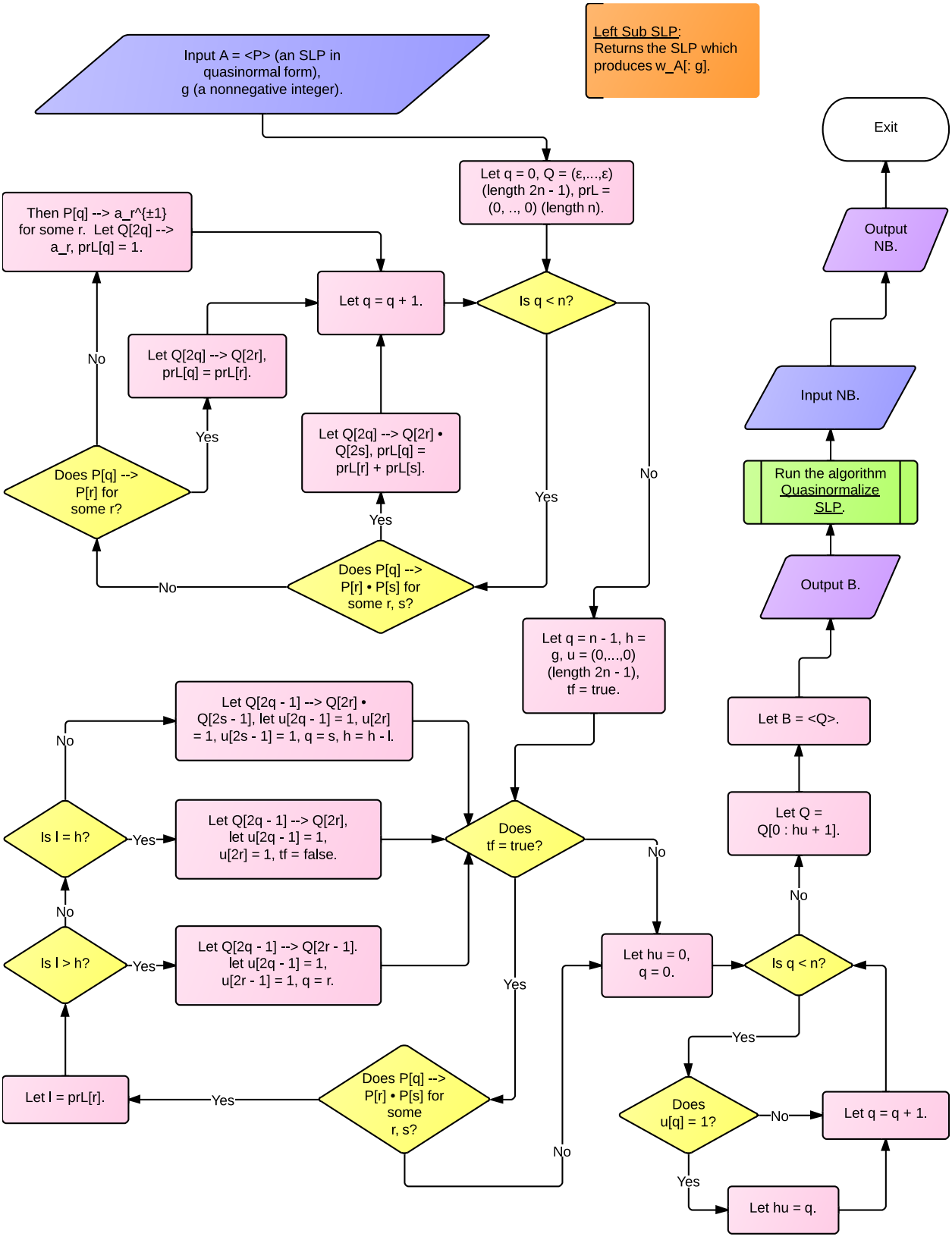


Figure 2.12: Algorithm Left Sub SLP

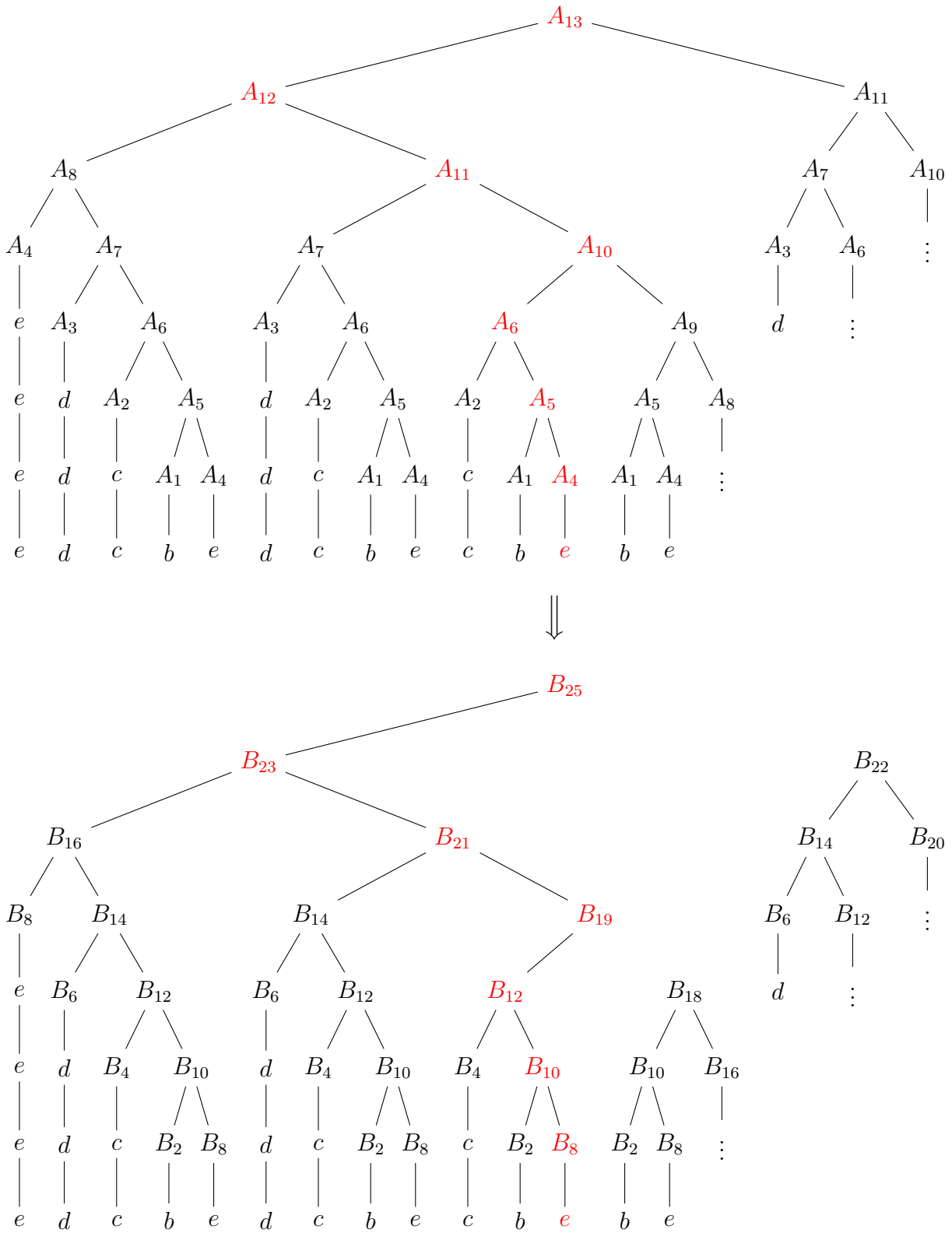


Figure 2.13: Producing a Left Subword of Length 12

construction, $\mathbb{B}$ has length no more than $2n$. By Lemma 2.11, the number of steps required to run Quasinormalize SLP on $\mathbb{B}$ is a polynomial in $2n$, say $p(2n)$. Now $p(2n)$ is a polynomial in n, so when we add $q(n) + 4c_2n + c_1 - c_2$ to $p(2n)$, we get a new polynomial in n. This is the time required to run the algorithm Left Sub SLP. □

Lemma 2.24. *The length of the SLP produced by Left Sub SLP is no more than $2n$, where n is the length of the SLP which is input.*

Proof. If n is the length of the SLP which is input, then by construction, $\mathbb{B}$ has length no more than $2n$. By Lemma 2.12, applying Quasinormalize SLP to $\mathbb{B}$ produces an SLP which is no longer than the length of $\mathbb{B}$. Thus the SLP output by Left Sub SLP has length no more than $2n$. □

Lemma 2.25. *The algorithm Right Sub SLP described by the flowchart in Figure 2.14 inputs an SLP $\mathbb{A}$ and a nonnegative integer f, and returns an SLP in quasinormal form which produces $w_A[L - f :]$, the rightmost subword of length f, where $\mathbb{A}$ produces the word w_A, which has length L.*

Proof. The algorithm Right Sub SLP is almost exactly the same as the algorithm Left Sub SLP, so we will merely discuss the differences, all of which are immediately before or inside the main loop. The name of the integer we input in Right Sub SLP is f, so we set h to f rather than to g, as we did in Left Sub SLP. Because we want the rightmost subword now, we set l to $\mathbf{prL}[s]$, the length of $w(A_s)$, where we set l to $\mathbf{prL}[r]$, the length of $w(A_r)$, in Left Sub SLP. We still have the three possibilities of $l > h$, $l = h$, or $l < h$.

If $l > h$, we now set $\mathbf{Q}[2q - 1]$ to $\mathbf{Q}[2s - 1]$ rather than to $\mathbf{Q}[2r - 1]$, and q to s, not r. Additionally, we set $\mathbf{u}[2q - 1]$ and $\mathbf{u}[2s - 1]$ to 1. In this way, the production rule for A_r will not be used, and the one for A_s will be replaced with a production which produces a shorter word. We then return to the beginning of the main loop.

If $l = h$, we set $\mathbf{Q}[2q-1]$ to $\mathbf{Q}[2s]$ instead of to $\mathbf{Q}[2r]$, and we set $\mathbf{u}[2q-1]$ and $\mathbf{u}[2s]$ to 1. As in Left Sub SLP, we then exit the main loop.

In the case where $l < h$, instead of setting $\mathbf{Q}[2q-1]$ to $\mathbf{Q}[2r] \cdot \mathbf{Q}[2s-1]$, we set it to $\mathbf{Q}[2r-1] \cdot \mathbf{Q}[2s]$, and we set q to r instead of to s. We also set $\mathbf{u}[2q-1]$, $\mathbf{u}[2r-1]$, and $\mathbf{u}[2s]$ to 1, and we set the value of h to $h-l$. This results in the production rule for A_s being kept, while the one for A_r will be considered next. We then return to the beginning of the loop.

The result of these changes is that we are now keeping and, when necessary, modifying the production rules which produce the rightmost subwords of w_A, rather than those which produce the leftmost subwords. The length of the produced subword is f by construction. □

Lemma 2.26. *The algorithm Right Sub SLP described by the flowchart in Figure 2.14 runs in polynomial time in n, the length of the SLP, $\mathbb{A}$, which is initially input.*

Proof. The differences between Right Sub SLP and Left Sub SLP do not affect the number of steps involved in running the routines. Therefore, since Left Sub SLP runs in polynomial time in n by Lemma 2.23, so does Right Sub SLP. □

Lemma 2.27. *The length of the SLP produced by Right Sub SLP is no more than $2n$, where n is the length of the SLP which is input.*

Proof. If n is the length of the SLP which is input, then by construction, the produced SLP has length no more than $2n$. By Lemma 2.12, applying Quasinormalize SLP to that SLP does not increase the length of the SLP. Thus the SLP output by Right Sub SLP has length no more than $2n$. □

Lemma 2.28. *The algorithm Find the Occurrence described by the flowchart in Figure 2.15 inputs an SLP $\mathbb{A}$ and integers p and u with $p > -2$, $p \neq 0$, and $u > 0$, and returns the position of the p^{th} occurrence of $a_u^{\pm 1}$ in the word produced by $\mathbb{A}$, where $p = -1$ indicates the last (rightmost) occurrence.*

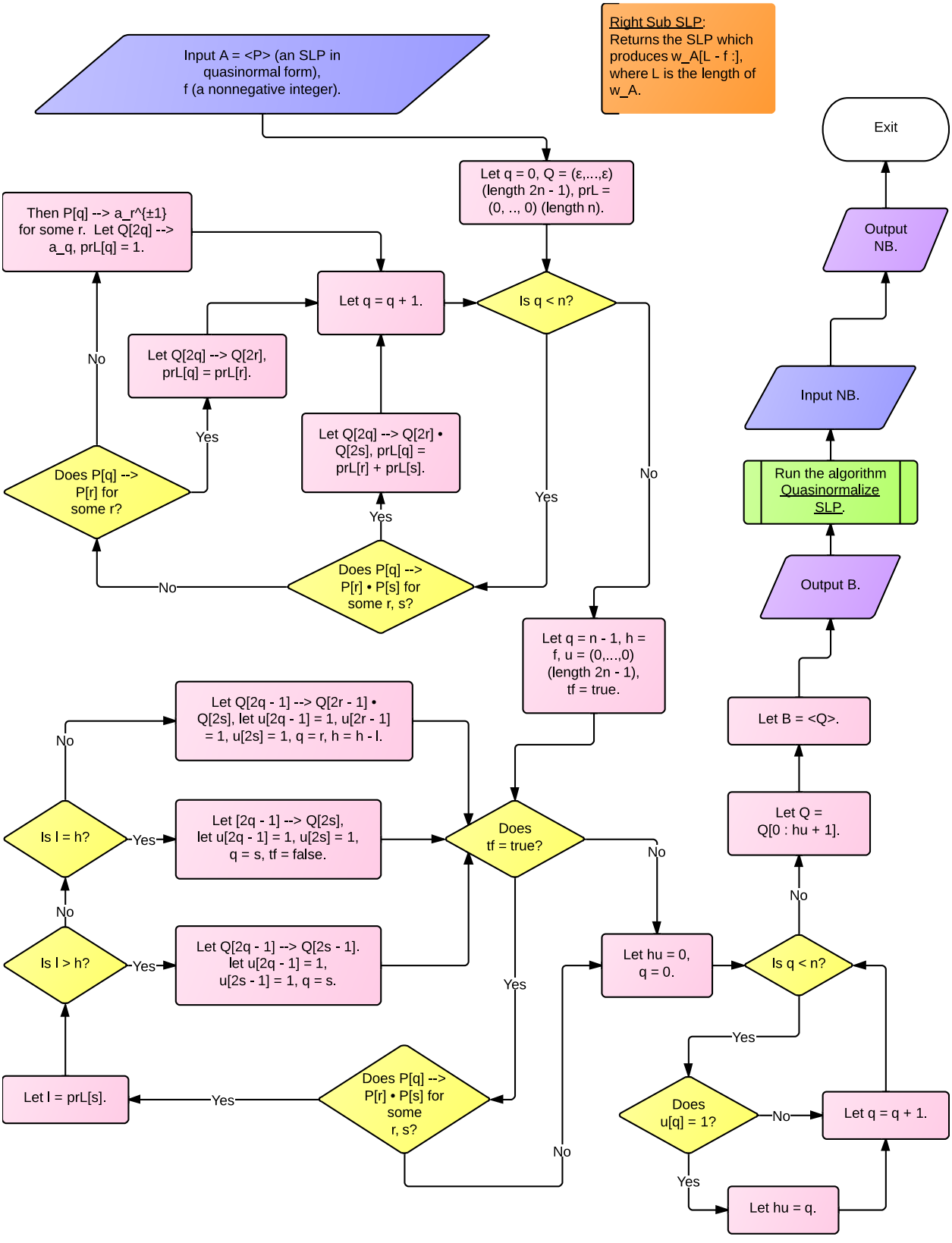


Figure 2.14: Algorithm Right Sub SLP

Proof. We first output $\mathbb{A}$ and u to the algorithm Count the Occurrences, which returns the integer k. By Lemma 2.18, k is the number of occurrences of $a_u^{\pm 1}$ in w_A. We check to see if $p > k$, and if so, we return -1, indicating that there are not p occurrences of $a_u^{\pm 1}$ in w_A, and exit the program. If not, we check to see if $p = -1$, and if so, we change p to k so that the p^{th} occurrence of $a_u^{\pm 1}$ is the last, or rightmost, occurrence.

Regardless of whether or not p was originally -1, we proceed now by creating two lists: **occs** and **prL**, both of length n, with all list items set to 0. Each item in the two lists corresponds to a non-terminal character. We use **occs** to store the number of occurrences of $a_u^{\pm 1}$ in the words produced by each production rule, and we use **prL** to store the lengths of the words produced by each production rule.

We begin by stepping through the production rules of $\mathbb{A}$ one at a time using the index q, starting with $q = 0$. In the first loop we find and enter the correct values for each item in these two lists. Consider the following three cases. If the current production rule of $\mathbb{A}$ is $A_q \to a_r^{\pm 1}$ for some r, we set $\mathbf{prL}[q]$ to 1. We check to see if $r = u$, and if so, we set $\mathbf{occs}[q]$ to 1; if not, we leave $\mathbf{occs}[q]$ set to 0. In the second case, the rule is of the form $A_q \to A_r$, and we set $\mathbf{prL}[q]$ to $\mathbf{prL}[r]$ and $\mathbf{occs}[q]$ to $\mathbf{occs}[r]$. In the third case, the rule is $A_q \to A_r \cdot A_s$ for some r and s, and we set $\mathbf{prL}[q]$ to $\mathbf{prL}[r] + \mathbf{prL}[s]$ and $\mathbf{occs}[q]$ to $\mathbf{occs}[r] + \mathbf{occs}[s]$. Thus for any q, $\mathbf{prL}[q]$ is the length of $w(A_q)$, and $\mathbf{occs}[q]$ is the number of occurrences of $a_u^{\pm 1}$ in $w(A_q)$.

To begin the main loop we set q to $n - 1$ and initialize the variables j and tf to 0 and true, respectively. We use j to keep track of the position of the rightmost letter which has been determined to be left of the p^{th} occurrence of $a_u^{\pm 1}$, and we use tf to allow us to break out of the main loop at the right time. Now we step through those production rules corresponding to characters which lie along the branch leading to the p^{th} occurrence of $a_u^{\pm 1}$. Beginning with $q = n - 1$, we consider the corresponding production rule of $\mathbb{A}$. As we cycle through this main loop, q becomes smaller and smaller. Once we reach the first production rule which is

not of the form $A_q \to A_r \cdot A_s$, we break out of the main loop and proceed to the next step. So all of the production rules considered in the main loop, except for the last one, are of the form $A_q \to A_r \cdot A_s$. For each of these encountered, we set the variable v to $\mathbf{occs}[r]$, the number of occurrences of $a_u^{\pm 1}$ in $w(A_r)$. There are two possibilities: $v < p$ or $v \geq p$.

If $v < p$, this means that the word produced by A_r does not contain the p^{th} occurrence of $a_u^{\pm 1}$, so the word produced by A_s must contain it. We set p to $p - v$ because the first v occurrences we need lie in $w(A_r)$, so in the next step we will need only $p - v$ more occurrences. We set q to s because we need to consider the production rule for A_s in the next run through the loop, and we set j to $j + \mathbf{prL}[q]$ since the p^{th} occurrence must lie to the right of the rightmost letter of $w(A_r)$. We then return to the beginning of the main loop.

If $v \geq p$, this means that $w(A_r)$ contains the p^{th} occurrence of $a_u^{\pm 1}$, so we need to consider the production rule for A_r in the next run through the loop. To do so, we set q to r and leave p and j unchanged. We then return to the beginning of the main loop.

Once we reach the first production rule not of the form $A_q \to A_r \cdot A_s$, we set j to $j + 1$, because by construction of the algorithm, $w(A_q)$ must be the p^{th} occurrence of $a_u^{\pm 1}$. Before this step, j was set to the position of the letter to the immediate left of $w(A_q)$, so we add 1 to make j the position of the letter $w(A_q)$ in w_A. We then set tf to false, causing the main loop to terminate.

Now the loop must terminate; we will eventually encounter a production rule which is not $A_q \to A_r \cdot A_s$ for some r and s. This happens because at each step where the production rule is $A_q \to A_r \cdot A_s$ for some r and s, the value of q is changed to r or s for the next iteration of the loop, and both r and s must be less than the current value of q, because $\mathbb{A}$ is in quasinormal form. So each time the loop repeats, the value of q is less than the previous time. Additionally, the production rules for terminal characters have smaller indices than those for non-terminal characters, so at some point q will be small enough that the associated

production rule will be for a terminal character, causing the loop to terminate.

Once outside the main loop, we merely return j and exit the program. □

Lemma 2.29. *The algorithm Find the Occurrence described by the flowchart in Figure 2.15 runs in polynomial time in n, the length of the SLP, $\mathbb{A}$, which is initially input.*

Proof. The time it takes to run the algorithm Count the Occurrences is polynomial in n, say $p(n)$, by Lemma 2.19. The other operations that happen outside the two loops, as well as the time required to determine whether or not $q < n$ or $tf = true$, do not depend on n and so happen in constant time, say c_1 steps. For each iteration of each of the two loops, the time is independent of n, so the number of steps is bounded by a constant; let the maximum of these two constants be c_2. The first loop runs n times, and the second loop runs no more than n times, so the algorithm Find the Occurrence requires no more than $p(n) + c_1 + c_2(n + n)$ steps, which is a polynomial in n. □

The next two algorithms we discuss involve checking to see whether or not the rightmost subword of one word is equal to the inverse of the leftmost subword of another word, and if so, canceling those subwords. We use these routines when trying to put words into shortest form. If the rightmost subword of length r of a word w_A is the same as the inverse of the leftmost subword of length r of a word w_B, then the word $w_A \cdot w_B$ is equivalent as a group element to the word $w_A[: L - r] \cdot w_B[r :]$, where L is the length of w_A, so we perform this cancelation to get a shorter form of the element. It is worth noting that the algorithm Do They Cancel? is the one which uses Plandowski's algorithm. (Our actual Python routine does not use Plandowski's algorithm, but the algorithm indicated by the flowchart does use it.)

Lemma 2.30. *The algorithm Do They Cancel? described by the flowchart in Figure 2.16 inputs two SLPs, $\mathbb{A}$ and $\mathbb{B}$, and a nonnegative integer r, and returns a value of true if the rightmost subword of w_A of length r is the inverse of the leftmost subword of w_B of length r and a value*

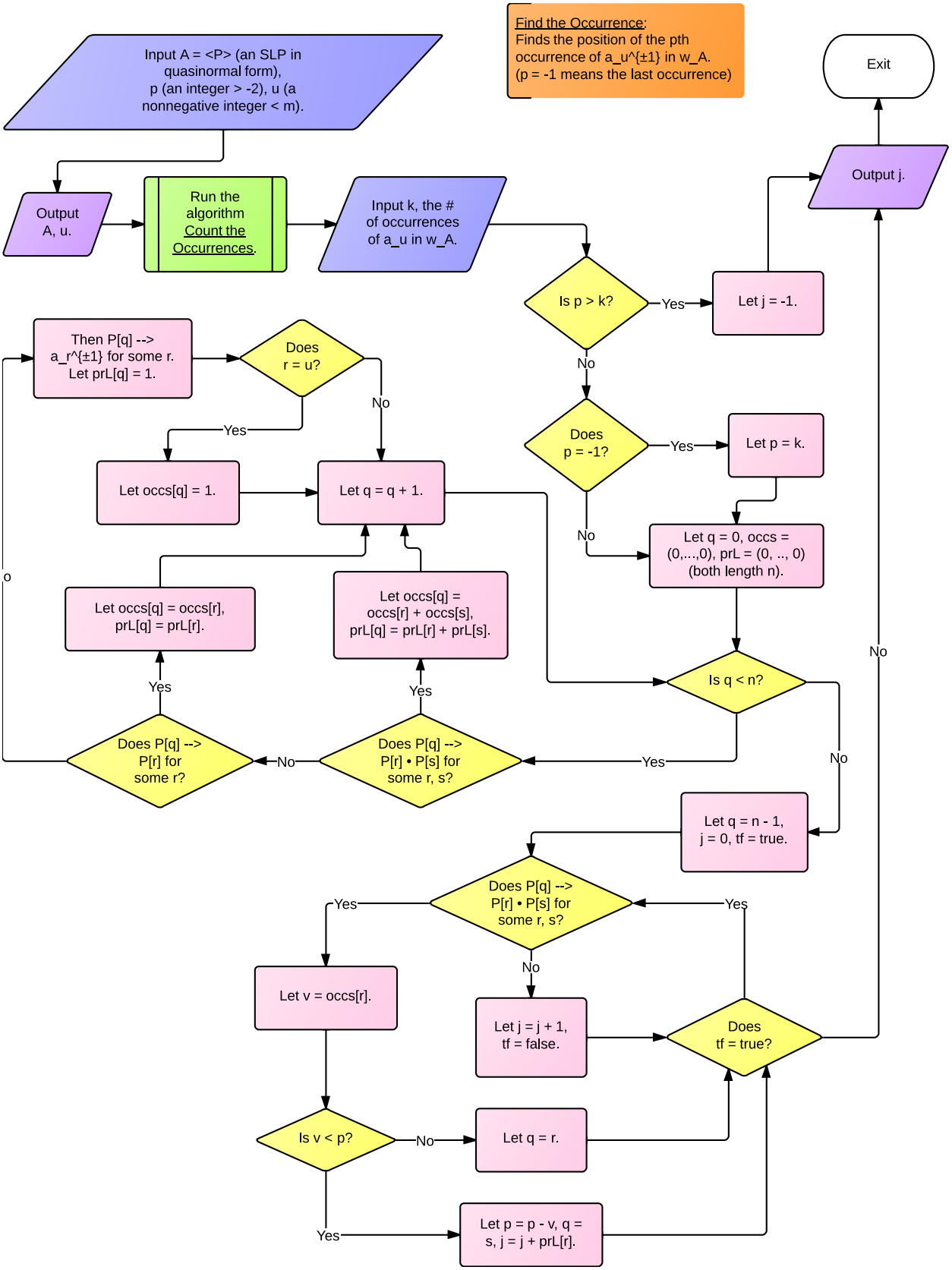


Figure 2.15: Algorithm Find the Occurrence

of false otherwise.

Proof. We first output $\mathbb{A}$ and r to the algorithm Right Sub SLP, which by Lemma 2.25 returns an SLP $\mathbb{C}$ which produces the rightmost subword of w_A of length r, that is, $w_A[L-r:]$, where L is the length of w_A.

We next output $\mathbb{B}$ and r to the algorithm Left Sub SLP, which by Lemma 2.22 returns an SLP $\mathbb{D}$ which produces the leftmost subword of w_B of length r, that is, $w_B[:r]$. Then we output $\mathbb{D}$ to the algorithm Inverse SLP, which returns an SLP $\mathbb{E}$. By Lemma 2.14, the word produced by $\mathbb{E}$ is the inverse of the word produced by $\mathbb{D}$, so $w_E = (w_B[:r])^{-1}$.

Finally, we output $\mathbb{C}$ and $\mathbb{E}$ to Plandowski's algorithm, which by [7] returns a value of true if $w_C = w_E$ and false otherwise. We input this value, h, and then return h and exit the routine. Thus we return true if $w_A[L-r:] = (w_B : r])^{-1}$ and false otherwise. □

Lemma 2.31. *The algorithm Do They Cancel? described by the flowchart in Figure 2.16 runs in polynomial time in $n+p$, where n and p are the lengths of the SLPs, $\mathbb{A}$ and $\mathbb{B}$, which are initially input.*

Proof. There are no loops in this routine, just calls to four other routines and then outputting the value h at the end. The time required to output information to and input information from other routines is bounded by a constant, so say it takes c steps for all of the inputting and outputting done in Do They Cancel?. By Lemmas 2.26, 2.23, and 2.15, the algorithms Right Sub SLP, Left Sub SLP, and Inverse SLP all run in polynomial time in the size of the SLPs which are input. So Right Sub SLP runs in polynomial time in n, say $q(n)$, and Left Sub SLP runs in polynomial time in p, say $s(p)$. The length of $\mathbb{C}$ must be polynomial in n, say $t(n)$, since it is produced by Left Sub SLP, and the length of $\mathbb{D}$ is polynomial in p, say $u(p)$. Thus Inverse SLP runs in polynomial time in $u(p)$, say $v(u(p))$, but composing two polynomials results in a new polynomial, so $v(u(p)) = f(p)$ for some polynomial f; hence

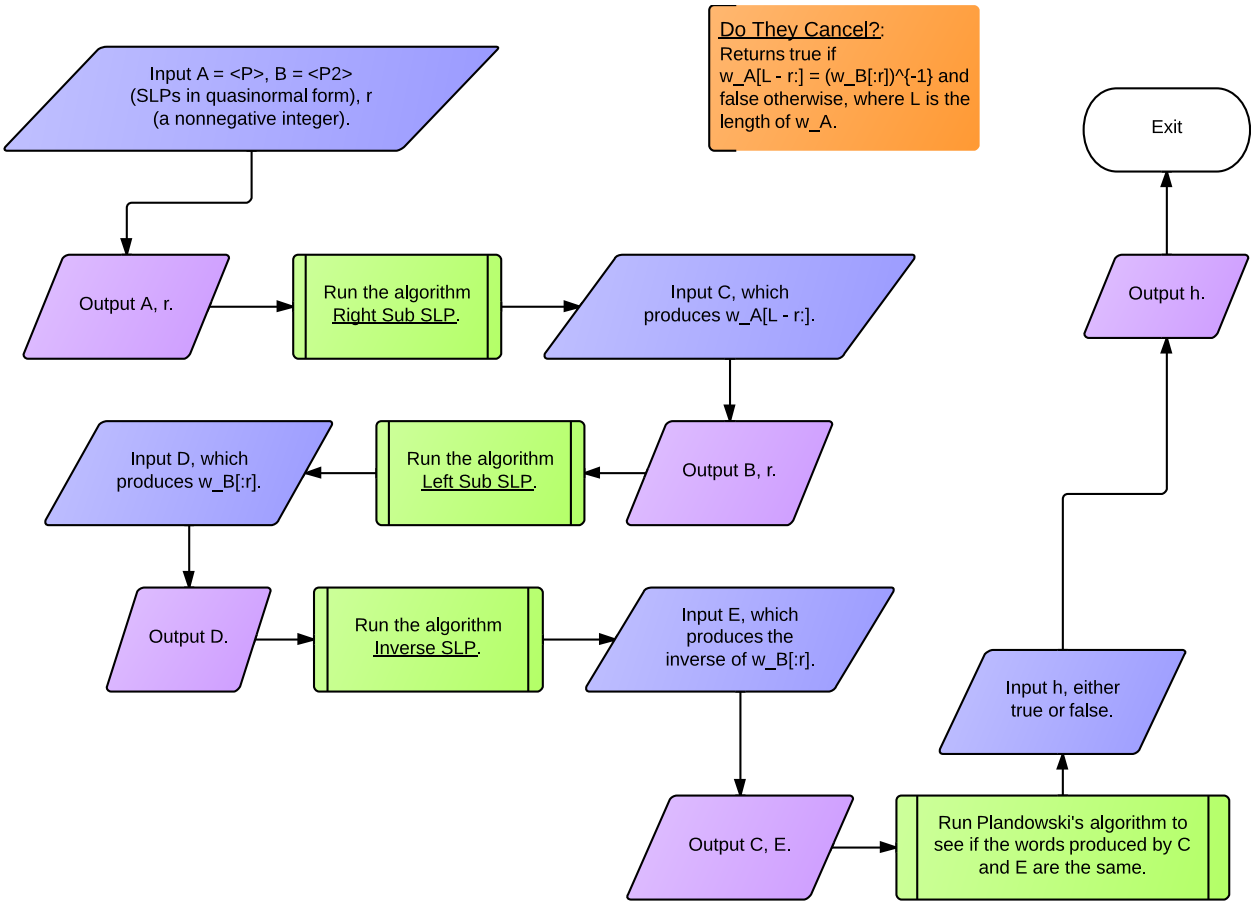


Figure 2.16: Algorithm Do They Cancel?

Inverse SLP runs in polynomial time in p. Inverse SLP produces $\mathbb{E}$, which therefore has polynomial length, say $g(p)$.

We output $\mathbb{C}$, which has length $t(n)$, and $\mathbb{E}$, which has length $g(p)$, into Plandowski's algorithm. By [7], Plandowski's algorithm runs in polynomial time in $t(n) + g(p)$, which is polynomial in $n + p$, say $F(n + p)$. If the polynomial indicating the number of steps required by Plandowski's algorithm is P, then the number of steps taken to run on $\mathbb{C}$ and $\mathbb{E}$ is $P(F(n + p))$, which is a new polynomial in $n + p$, say $Q(n + p)$. Therefore the number of steps required to run the algorithm Do They Cancel? is $c + q(n) + s(p) + f(p) + Q(n + p)$, which is again polynomial in $n + p$. □

The following routine performs the cancelation which Do They Cancel? checks for. The words produced by the SLPs which are returned are the same as the words produced by the SLPs which are input, but with the rightmost subword of the first and the leftmost subword of the second truncated by the same amount. The number of letters to truncate is not the integer which is input, however; the position of the leftmost letter to truncate in the first word is the integer which is input. Understanding this may help clarify the statement of the following lemma.

Lemma 2.32. *The algorithm Cancel Them described by the flowchart in Figure 2.17 inputs two SLPs, $\mathbb{A}$ and $\mathbb{B}$, and a nonnegative integer s, and returns two SLPs $\mathbb{ND}$ and $\mathbb{NE}$ in quasinormal form meeting the following conditions. The SLP $\mathbb{ND}$ produces the leftmost subword of w_A of length $s - 1$; that is, $w_{ND} = w_A[: s - 1]$. If the length of w_A is k_1, then the number of letters which occur in w_A but not in w_{ND}, that is, the number of letters cut from w_A to get w_{ND}, is $k_1 - (s - 1)$. This is also the number of letters cut from w_B to get w_{NE}. If k_2 is the length of w_B, then $\mathbb{NE}$ produces the rightmost subword of w_B of length $k_2 - (k_1 - (s - 1))$; that is, $w_{NE} = w_B[k_1 - (s - 1) :]$.*

Proof. We first output $\mathbb{A}$ and then $\mathbb{B}$ to the algorithm Get Length, which by Lemma 2.4

returns k_1 and then k_2, the lengths of w_A and w_B, respectively. We assign to the variable k the value $k_2 - (k_1 - (s-1)) = k_2 - k_1 + s - 1$ because $k_1 - (s-1)$ is the length of the subword that was truncated from w_A to get w_D, and we want to truncate the same number of letters from w_B. If $s > k_1$ or $k > k_2$ then an error message is printed to the screen and $\mathbb{A}$ and $\mathbb{B}$ are returned as the routine is exited; otherwise we proceed with the rest of the routine.

We next output $\mathbb{A}$ and $s-1$ to the algorithm Left Sub SLP, which by Lemma 2.22 returns an SLP $\mathbb{D}$ which produces the leftmost subword of w_A of length $s-1$, that is, $w_D = w_A[:s-1]$. We want the length of the subword of w_B that we keep to be $k = k_2 - (k_1 - (s-1))$, and so we output $\mathbb{B}$ and k to the algorithm Right Sub SLP, which returns an SLP $\mathbb{E}$. By Lemma 2.25, the word produced by $\mathbb{E}$ is the rightmost subword of $\mathbb{B}$ of length k. In other words, $w_E = w_B[k_1 - (s-1) :]$.

Now w_D and w_E produce the words we want, but they might not be in quasinormal form, so we output $\mathbb{D}$ and then $\mathbb{E}$ to the routine Quasinormalize SLP, which returns first $\mathbb{ND}$ and then $\mathbb{NE}$. By Lemma 2.10, $\mathbb{ND}$ and $\mathbb{NE}$ are SLPs in quasinormal form which produce the same words as $\mathbb{D}$ and $\mathbb{E}$, respectively. We return $\mathbb{ND}$ and $\mathbb{NE}$ and exit the routine. □

Lemma 2.33. *The algorithm Cancel Them described by the flowchart in Figure 2.17 runs in polynomial time in $n + p$, where n and p are the lengths of the SLPs, $\mathbb{A}$ and $\mathbb{B}$, which are initially input. Furthermore, the sizes of the SLPs which are output are no more than $2n$ and $2p$, respectively.*

Proof. There are no loops in this routine, just calls to four other routines, one calculation, and then outputting the new SLPs at the end. The time required to output information to and input information from other routines is bounded by a constant, as is the time to perform the one calculation and assign the result to a variable, so say it takes c steps for all of the steps done outside of other routines in Cancel Them. By Lemma 2.5, the algorithm Get Length runs in polynomial time, say $Q(x)$, in the size of the SLP which is input. So the

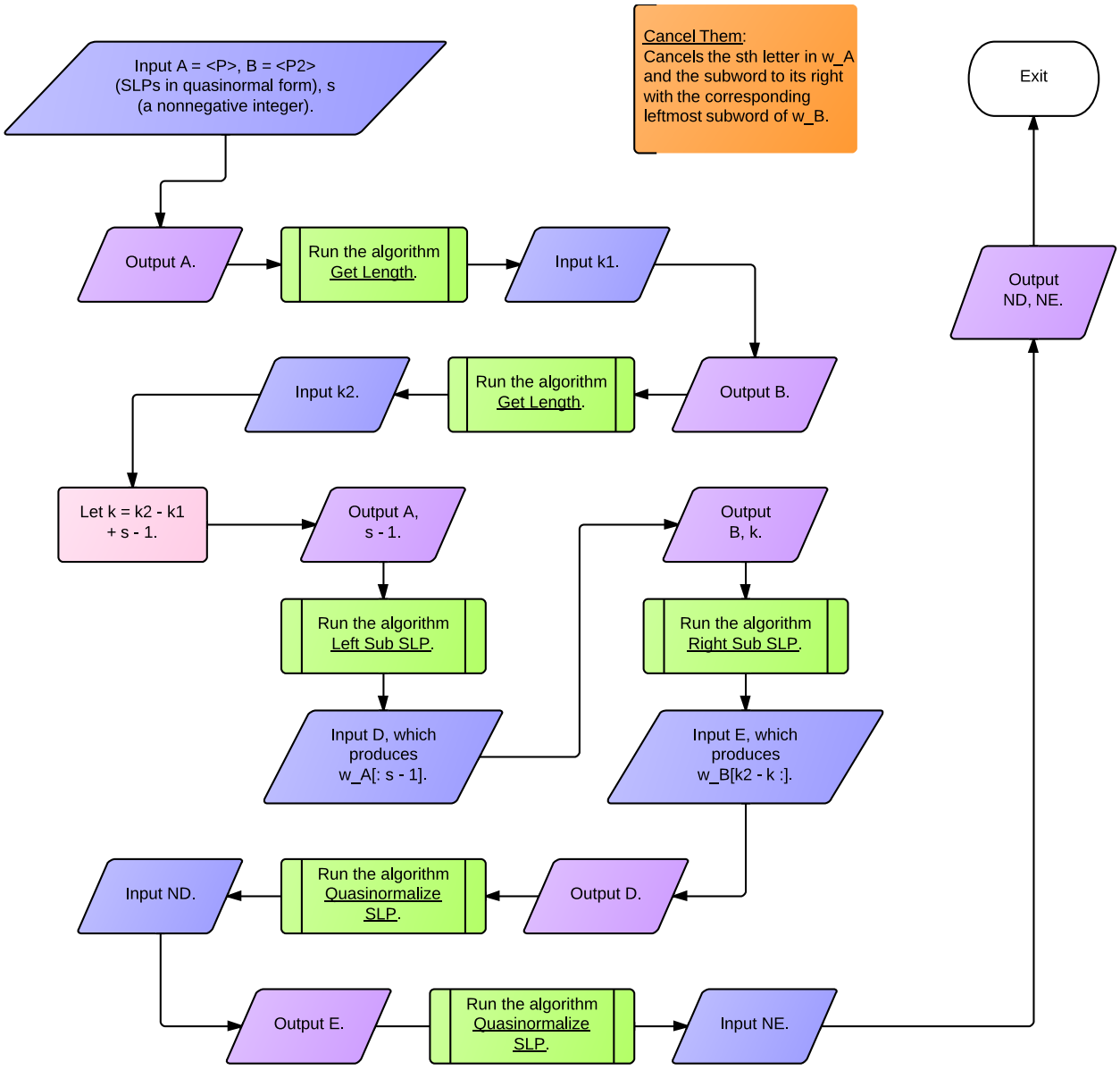


Figure 2.17: Algorithm Cancel Them

first time Get Length is called in Cancel Them, it runs in $Q(n)$ steps, and the second time it runs in $Q(p)$ steps.

By Lemmas 2.26 and 2.23, the algorithms Right Sub SLP and Left Sub SLP run in polynomial time in the size of the SLPs which are input. So in Cancel Them, Left Sub SLP runs in polynomial time in n, say $q(n)$, and Right Sub SLP runs in polynomial time in p, say $s(p)$. The length of $\mathbb{D}$ must be no more than $2n$, since it is produced by Left Sub SLP, and the length of $\mathbb{E}$ is at most $2p$, since it is produced by Right Sub SLP, by Lemmas 2.24 and 2.27.

Finally we quasinormalize each of these new SLPs. By Lemma 2.11, Quasinormalize SLP runs in polynomial time, say $v(x)$, in the size of the input. When $\mathbb{D}$ is input, then, Quasinormalize SLP requires $v(2n)$ steps, and when $\mathbb{E}$ is input, Quasinormalize SLP requires $v(2p)$ steps. Therefore the number of steps required to run the algorithm Cancel Them is $c + Q(n) + Q(p) + q(n) + s(p) + v(2n) + v(2p)$, which is itself bounded by a polynomial in $n + p$.

Furthermore, since the size of $\mathbb{D}$ is at most $2n$ and the size of $\mathbb{E}$ is at most $2p$, and since by Lemma 2.12, Quasinormalize SLP produces an SLP no bigger than that which was input, the length of $\mathbb{ND}$ is no more than $2n$, and the length of $\mathbb{NE}$ is no more than $2p$. □

2.3 Algorithms, Part 2 – Preliminaries for Ordering Lexicographically

The next several algorithms we discuss involve finding the position or positions of a letter or letters with certain attributes, such as the leftmost letter in a word which does not commute with a particular list of generators. These are all used when putting a word into lexicographic order. Recall that we do not consider a generator able to commute with itself or its inverse, and that the global list **R** contains all pairs of indices corresponding to pairs of generators which commute with each other.

Lemma 2.34. *The algorithm Find Rightmost Noncommuting described by the flowchart in Figure 2.18 inputs an SLP $\mathbb{A}$ and a nonnegative integer $i < m$ and returns the index h and position t of the rightmost letter $a_h^{\pm 1}$ in w_A which does not commute with a_i.*

Proof. We input an SLP $\mathbb{A}$, and a nonnegative integer $i < m$. We output $\mathbb{A}$ to the algorithm Find Included Generators which, by Lemma 2.6, returns a list $\mathbf{y}$ which has each item $\mathbf{y}[j]$ set to 1 if $a_j^{\pm 1}$ occurs in w_A and set to 0 otherwise. In the first loop we modify $\mathbf{y}$ by changing $\mathbf{y}[j]$ to 0 if it had been 1 and if a_j commutes with a_i. Then the only items in $\mathbf{y}$ set to 1 will be those for which $a_j^{\pm 1}$ occurs in w_A and a_j does not commute with a_i. We use the index j to step through the items of $\mathbf{y}$ one at a time, starting with $j = 0$. For each j we check to see if $\mathbf{y}[j] = 1$; if not, we return to the beginning of the loop. For each $\mathbf{y}[j]$ set to 1 we check to see if the pair (i, j) or the pair (j, i) is in $\mathbf{R}$. If either pair is in $\mathbf{R}$, we change $\mathbf{y}[j]$ to 0; otherwise we leave $\mathbf{y}[j]$ unchanged. We then return to the beginning of the loop. We exit this loop when $j = m$, the length of $\mathbf{y}$.

We initialize the variables h and t to -1 and reset j to 0 before beginning the main loop. In the main loop, we first check whether or not $\mathbf{y}[j] = 1$; if not, we return to the beginning of the loop. For each $\mathbf{y}[j]$ set to 1 we output $\mathbb{A}$, -1, and j to the algorithm Find the Occurrence, which returns the position s of the rightmost occurrence of $a_j^{\pm 1}$, by Lemma 2.28. If s is greater than the current value of t, we set t to s and h to j; otherwise we leave h and t unchanged. We then return to the beginning of the main loop. Since we only run Find the Occurrence for those values of j for which a_j and a_i do not commute, every value of s which is returned is the position of a letter which does not commute with a_i. And since we run Find the Occurrence for every value of j for which $a_j^{\pm 1}$ occurs in w_A and a_j does not commute with a_i, the rightmost position of every letter in w_A which does not commute with a_i is given by s at some point during the main loop. Now we only change the value of t to be that of s if $s > t$, so at the end of the loop, unless t and h are still set to their initial values, t is the position of the rightmost letter in w_A which does not commute with a_i. And because we

change the value of h to that of j only when we change t to be that of s, h is the index of that letter. If t and h are both still -1, this indicates that w_A contains no letter which does not commute with a_i. We return h and t. □

Lemma 2.35. *The algorithm Find Rightmost Noncommuting described by the flowchart in Figure 2.18 runs in polynomial time in n, the length of the SLP which is input.*

Proof. The time it takes to output information to and input information from other routines, as well as the time required to determine whether or not $j < m$, do not depend on n and so happen in constant time, say c_1 steps. The algorithm Find Included Generators runs in polynomial time in n, say $p(n)$ steps, by Lemma 2.7. For each iteration of the first loop, the time is independent of n, and so the number of steps is bounded by a constant, say c_2. For each iteration of the main loop, the number of steps outside of running Find the Occurrence is bounded by a constant, say c_3. By Lemma 2.29, Find the Occurrence runs in polynomial time in n, say $q(n)$. Each of the two loops runs m times, so the number of steps required to run Find Rightmost Noncommuting is $c_1 + p(n) + c_2 m + m(c_3 + q(n))$, which is a polynomial in n. □

The routine we discuss next is very similar to the previous one, but instead of finding the rightmost occurrence it finds the leftmost, and instead of only considering those letters which do not commute with a particular generator, it considers all letters which fail to commute with at least one generator in a list of indices of generators. For the sake of brevity, let us refer to the letters whose indices appear in that list, **iList**, as **iList** *letters.*

Lemma 2.36. *The algorithm Find Leftmost Noncommuting - List described by the flowchart in Figure 2.19 inputs an SLP $\mathbb{A}$ and a list* **iList** *of nonnegative integers less than m and returns the index h and position t of the leftmost letter $a_h^{\pm 1}$ in w_A which fails to commute with a_i for at least one i in* **iList**.

Proof. As in Find Rightmost Noncommuting, we begin by outputting $\mathbb{A}$ to the algorithm

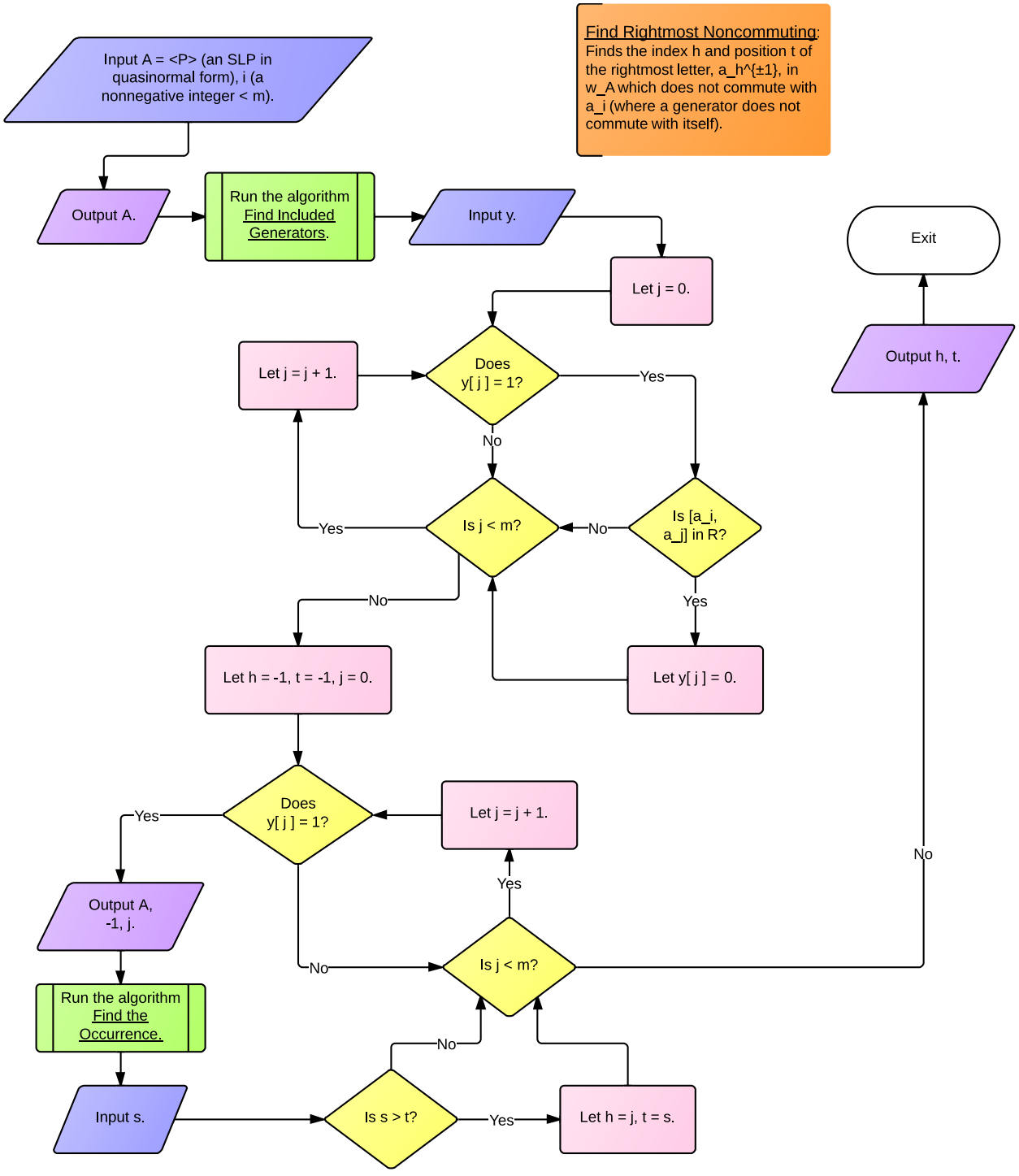


Figure 2.18: Algorithm Find Rightmost Noncommuting

Find Included Generators which, by Lemma 2.6, returns a list **y** which has each item **y**$[j]$ set to 1 if $a_j^{\pm 1}$ occurs in w_A and set to 0 otherwise. Before entering the first loop, we create a list **y2** of length m with each item set to 0, and we set the constant L to the length of **iList**. We use **y2** to indicate which generators fail to commute with at least one letter whose index is in **iList**. In the first loop we modify **y2** by changing **y2**$[j]$ to 1 if **y**$[j]$ is 1 and if a_j fails to commute with at least one **iList** letter. We use the index j to step through the items of **y** one at a time, starting with $j = 0$. For each j we check to see if **y**$[j] = 1$; if not, we return to the beginning of the loop. For each **y**$[j]$ set to 1 we step through each item in the list **iList** using the index i, which runs from 0 to $L - 1$. For each i, we check to see if the pair (**iList**$[i], j$) or the pair (j, **iList**$[i]$) is in **R**. If neither pair is in **R**, we change **y2**$[j]$ to 1 and exit the smaller nested loop; otherwise we return to the beginning of the smaller nested loop. When exiting the smaller loop, either **y2**$[j]$ has been set to 1 or a_j commutes with every **iList** letter, in which case **y2**$[j]$ is still 0. We then return to the beginning of the outer loop to check the next j. We exit this loop when $j = m$, the length of **y**. Therefore as we exit the first loop, each item **y2**$[j]$ is 0 if $a_j^{\pm 1}$ occurs in w_A and a_j commutes with every **iList** letter, and is 1 otherwise.

We next output $\mathbb{A}$ to the algorithm Get Length, which returns k, the length of w_A, by Lemma 2.4. We initialize the variables h to -1 and t to $k + 1$ and reset j to 0 before beginning the main loop. The main loop has three differences from the main loop in Find Rightmost Noncommuting: we look at the list **y2** instead of the list **y**, we output 1 instead of -1 as the second value to Find the Occurrence, and we change the values of h and t when $s < t$ rather than when $s > t$. To begin the main loop, we first check whether or not **y2**$[j] = 1$; if not, we return to the beginning of the loop. For each **y2**$[j]$ set to 1 we output $\mathbb{A}$, 1, and j to the algorithm Find the Occurrence, which returns the position s of the leftmost occurrence of $a_j^{\pm 1}$, by Lemma 2.28. If s is less than the current value of t, we set t to s and h to j; otherwise we leave h and t unchanged. We then return to the beginning of the main loop. Since we only run Find the Occurrence for those values of j for which a_j fails to commute with at least

one **iList** letter, every value of s which is returned is the position of a letter which fails to commute with at least one **iList** letter. And since we run Find the Occurrence for every value of j for which $a_j^{\pm 1}$ occurs in w_A and a_j fails to commute with some **iList** letter, the leftmost position of every letter in w_A which does not commute with some **iList** letter is given by s at some point during the main loop. Now we only change the value of t to be that of s if $s < t$, so at the end of the loop, unless t and h are still set to their initial values, t is the position of the leftmost letter in w_A which does not commute with some **iList** letter. And because we change the value of h to that of j only when we change t to be that of s, h is the index of that letter. Just before exiting we check to see if $h = -1$, and if so, we change t to be -1, indicating that every letter in w_A commutes with all of the **iList** letters; there is no letter meeting the condition we want. Otherwise t is left unchanged. We return h and t. □

Lemma 2.37. *The algorithm Find Leftmost Noncommuting - List described by the flowchart in Figure 2.19 runs in polynomial time in n, the length of the SLP which is input.*

Proof. The time it takes for the operations outside of the called subroutines and the two loops does not depend on n, and so they happen in constant time, say c_1 steps. The algorithms Find Included Generators and Get Length run in polynomial time in n, say $p(n)$ steps total for the two routines, by Lemmas 2.7 and 2.5. For each iteration of the first loop, including running through the smaller nested loop up to L times, the time is independent of n, and so the number of steps is bounded by a constant, say c_2. For each iteration of the main loop, the number of steps outside of running Find the Occurrence is bounded by a constant, say c_3. By Lemma 2.29, Find the Occurrence runs in polynomial time in n, say $q(n)$. Each of the two loops runs m times, so the number of steps required to run Find Leftmost Noncommuting - List is $c_1 + p(n) + c_2 m + m(c_3 + q(n))$, which is a polynomial in n. □

The only difference between the routine just discussed and the one next discussed is that in the following one the list **y2** which is created has items **y2**$[j]$ set to 1 if a_j commutes

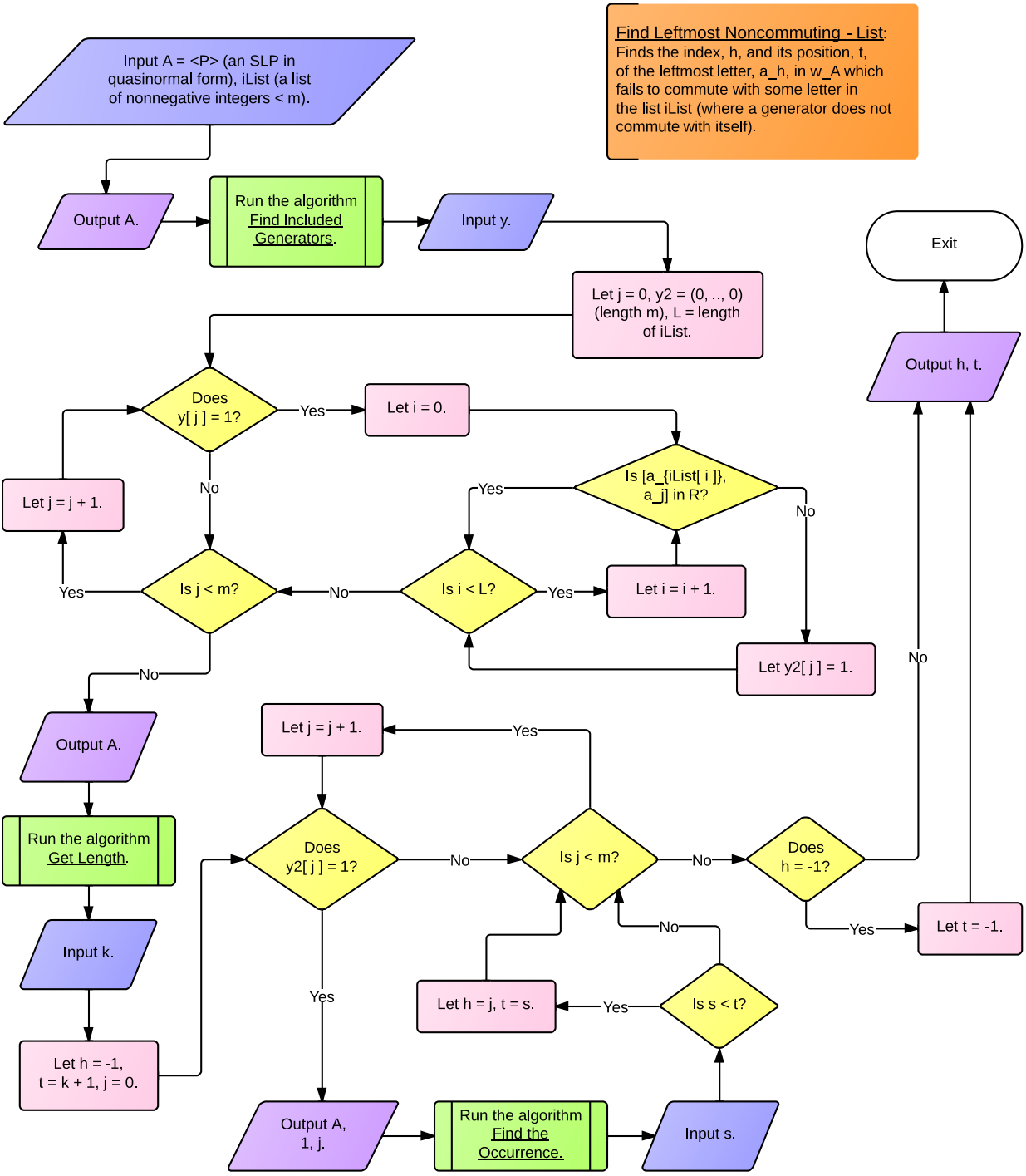


Figure 2.19: Algorithm Find Leftmost Noncommuting - List

with all letters whose indices are in **iList**. This results in Find Leftmost Commuting - List returning the index and position of the leftmost letter which commutes with all of the **iList** letters, rather than the leftmost letter which fails to commute with at least one of the **iList** letters.

Lemma 2.38. *The algorithm Find Leftmost Commuting - List described by the flowchart in Figure 2.20 inputs an SLP* $\mathbb{A}$ *and a list* **iList** *of nonnegative integers less than* m *and returns the index* h *and position* t *of the leftmost letter* $a_h^{\pm 1}$ *in* w_A *which commutes with* a_i *for every* i *in* **iList**.

Proof. Because Find Leftmost Commuting - List is so similar to Find Leftmost Noncommuting - List, we will merely discuss the differences and how they affect the outcome. In the algorithm Find Leftmost Commuting - List, the list **y2** of length m is initially set to be equal to **y**, rather than setting each item to 0. We then modify **y2** by changing $\mathbf{y2}[j]$ to 0 if $\mathbf{y}[j]$ is 1 and if a_j fails to commute with at least one **iList** letter. So inside the smaller nested loop, if neither the pair $(\mathbf{iList}[i], j)$ nor the pair $(j, \mathbf{iList}[i])$ is in $\mathbf{R}$ for some i, we change $\mathbf{y2}[j]$ to 0 and exit the smaller nested loop. Hence when exiting the smaller loop, either $\mathbf{y2}[j]$ has been set to 0 or a_j commutes with every **iList** letter, in which case $\mathbf{y2}[j]$ is still 1. Therefore as we exit the first loop, each item $\mathbf{y2}[j]$ is 1 if $a_j^{\pm 1}$ appears in w_A and a_j commutes with every **iList** letter, and is 0 otherwise.

The rest of the algorithm is exactly the same as in Find Leftmost Noncommuting - List. In Find Leftmost Commuting - List, we only run Find the Occurrence for those values of j for which a_j commutes with every **iList** letter, so every value of s which is returned is the position of a letter which commutes with every **iList** letter. And since we run Find the Occurrence for every value of j for which $a_j^{\pm 1}$ occurs in w_A and a_j commutes with all **iList** letters, the leftmost position of every letter in w_A which commutes with every **iList** letter is given by s at some point during the main loop. Since t is set to s whenever $s < t$, when we reach the end of the loop, t is the position of the leftmost letter in w_A which commutes with

all of the **iList** letters, and h is the index of that letter, unless t and h are still set to their initial values. Just before exiting we check to see if $h = -1$, and if so, we change t to be -1, indicating that every letter in w_A commutes with all of the **iList** letters; there is no letter meeting the condition we want. Otherwise t is left unchanged. We return h and t. □

Lemma 2.39. *The algorithm Find Leftmost Commuting - List described by the flowchart in Figure 2.20 runs in polynomial time in n, the length of the SLP which is input.*

Proof. The only difference between this algorithm and Find Leftmost Noncommuting - List is which items in **y2** get set to 0 and which get set to 1, which does not affect the time required to run the program. Therefore, since Find Leftmost Noncommuting - List runs in polynomial time in n by Lemma 2.37, so does Find Leftmost Commuting List. □

We discuss in the following lemma an algorithm very similar to the previous one. The differences are that an ordering on the generators is input at the beginning, and instead of finding the leftmost letter which commutes with a list, we find the leftmost letter which comes after a given letter in the ordering which is input. The ordering is input as a list **genord**, with each item **genord**$[j]$ set to the position of a_j in the ordering. For example, if **genord** is the list $[1, 3, 0, 2]$, this means that the generators a_0, a_1, a_2, and a_3 have been given the ordering a_2, a_0, a_3, a_1: Since **genord**$[2] = 0$, a_2 is in the 0^{th} position, or first in the ordering; since **genord**$[0] = 1$, a_0 is second; and so on. We often refer to this ordering as a weight. In the example just given, we say a_2 is *lighter than* a_0 and a_1 is *heavier than* a_3. By construction, a generator's inverse has the same weight as the generator, and two letters have the same weight if and only if they are inverses of each other or are equal.

Lemma 2.40. *The algorithm Find Leftmost Heavier described by the flowchart in Figure 2.21 inputs an SLP $\mathbb{A}$, a list* **genord** *of nonnegative integers less than m, and an integer $i < m$ and returns the index h and position t of the leftmost letter $a_h^{\pm 1}$ in w_A which is heavier than a_i in the ordering given by* **genord***; that is, such that* **genord**$[h] >$ **genord**$[i]$.

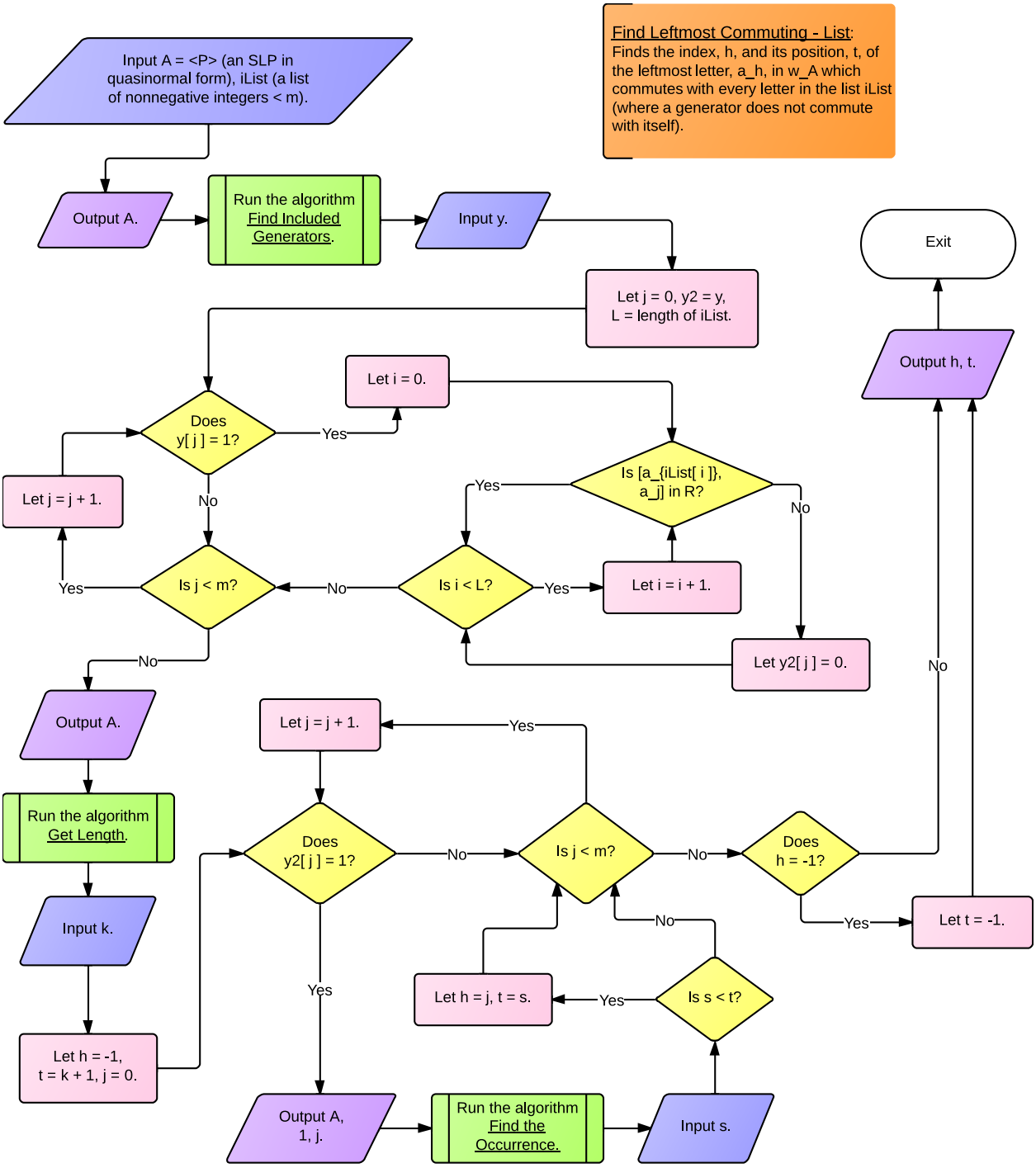


Figure 2.20: Algorithm Find Leftmost Commuting - List

Proof. Because Find Leftmost Heavier is so similar to the algorithm Find Leftmost Commuting - List, we will merely discuss the differences and how they affect the outcome. We input an SLP $\mathbb{A}$ as in Find Leftmost Commuting - List, but instead of the list **iList**, we input a list **genord** which gives an ordering of the generators and a nonnegative integer $i < m$. For any two generators a_k and a_l, a_k comes before a_l in the ordering if and only if $\mathbf{genord}[k] < \mathbf{genord}[l]$. In the algorithm Find Leftmost Heavier, the list $\mathbf{y}$ is modified itself, rather than being copied to another list which is then modified. We do this by changing $\mathbf{y}[j]$ to 0 if $\mathbf{y}[j]$ is 1 and if a_j is lighter than or equal in weight to a_i; that is, we set $\mathbf{y}[j]$ to 0 if and only if $\mathbf{y}[j]$ is 1 at the beginning of this iteration of the loop and $\mathbf{genord}[j] \leq \mathbf{genord}[i]$. Therefore as we exit the first loop, each item $\mathbf{y}[j]$ is 1 if $a_j^{\pm 1}$ appears in w_A and a_j is heavier than a_i, and is 0 otherwise.

The rest of the algorithm is exactly the same as in Find Leftmost Commuting - List, except that the list $\mathbf{y}$ is used instead of **y2**. In Find Leftmost Heavier, we only run Find the Occurrence for those values of j for which a_j is heavier than a_i, so every value of s which is returned is the position of a letter which is heavier than a_i. And since we run Find the Occurrence for every value of j for which $a_j^{\pm 1}$ occurs in w_A and a_j is heavier than a_i, the leftmost position of every letter in w_A heavier than a_i is given by s at some point during the main loop. Since t is set to s whenever $s < t$, when we reach at the end of the loop, unless t and h are still set to their initial values, t is the position of the leftmost letter in w_A which is heavier than a_i, and h is the index of that letter. Just before exiting we check to see if $h = -1$, and if so, we change t to be -1, indicating that every letter in w_A is lighter than or weighs the same as a_i; there is no letter meeting the condition we want. Otherwise t is left unchanged. We return h and t. □

Lemma 2.41. *The algorithm Find Leftmost Heavier described by the flowchart in Figure 2.21 runs in polynomial time in n, the length of the SLP which is input.*

Proof. The only differences between this algorithm and Find Leftmost Commuting List are

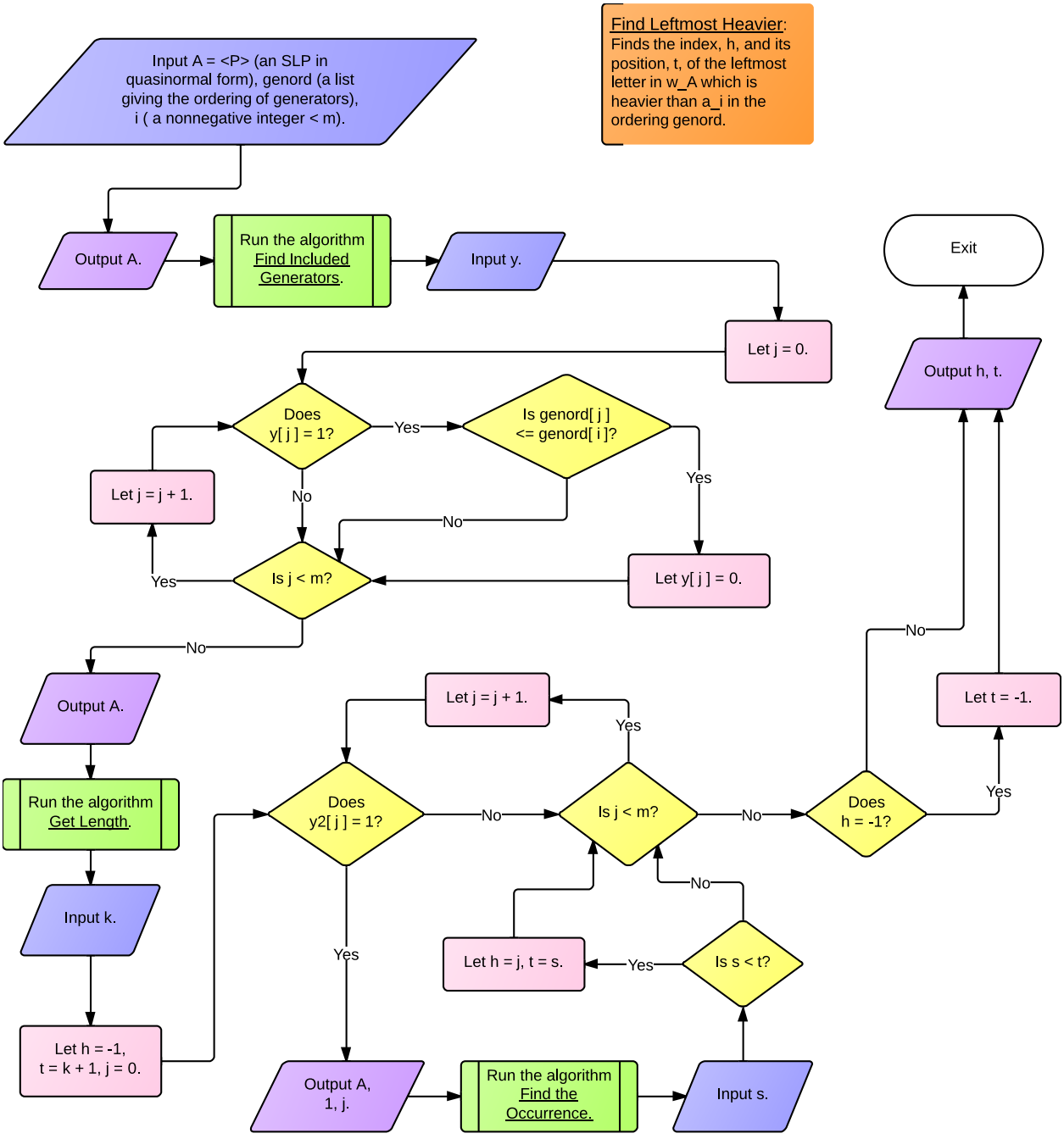


Figure 2.21: Algorithm Find Leftmost Heavier

that we use the list **y** itself rather than copying it to another list to use, there is no smaller loop nested inside the first loop, and which items of **y** get set to 0 and which get set to 1 is determined by comparing items in the list **genord** rather than checking for the occurrence of certain items in **R**. None of these differences affect the time required to run the program. Therefore, since Find Leftmost Commuting List runs in polynomial time in n by Lemma 2.39, so does Find Leftmost Heavier. □

The next two lemmas involve the algorithm Find Rightmost Heavier, which is exactly the same as Find Leftmost Heavier except that it finds the rightmost letter, rather than the leftmost letter, in w_A which is heavier than a_i in the ordering **genord** which is input, where $\mathbb{A}$ is the SLP which is input and i is the integer which is input.

Lemma 2.42. *The algorithm Find Rightmost Heavier described by the flowchart in Figure 2.22 inputs an SLP $\mathbb{A}$, a list* **genord** *of nonnegative integers less than m, and an integer $i < m$ and returns the index h and position t of the rightmost letter $a_h^{\pm 1}$ in w_A which is heavier than a_i in the ordering given by* **genord***; that is, such that* **genord**$[h] >$ **genord**$[i]$.

Proof. Because Find Rightmost Heavier is so similar to the algorithm Find Leftmost Heavier, we will merely discuss the differences and how they affect the outcome. The first loop is exactly the same as in Find Leftmost Heavier; thus by the proof of Lemma 2.40, as we exit the first loop, each item $\mathbf{y}[j]$ is 1 if $a_j^{\pm 1}$ appears in w_A and a_j is heavier than a_i, and is 0 otherwise.

The only differences are that we do not run the algorithm Get Length, we initialize t to 0 rather than to $k+1$, we output -1 rather than 1 as the second value to Find the Occurrence, and we do not check at the end to see if $h = -1$ or change t to -1. By Lemma 2.28, when we call the algorithm Find the Occurrence in this routine, the position s of the rightmost occurrence of a_j in w_A is returned. In Find Rightmost Heavier, we only run Find the Occurrence for those values of j for which a_j is heavier than a_i, so every value of s which

is returned is the position of a letter which is heavier than a_i. And since we run Find the Occurrence for every value of j for which $a_j^{\pm 1}$ occurs in w_A and a_j is heavier than a_i, the rightmost position of every letter in w_A heavier than a_i is given by s at some point during the main loop. Since t is set to s whenever $s > t$, when we reach the end of the main loop, t is the position of the rightmost letter in w_A which is heavier than a_i, and h is the index of that letter, unless t and h are still set to their initial values. If t and h are both still -1, this indicates that w_A contains no letter which is heavier than a_i. We return h and t. □

Lemma 2.43. *The algorithm Find Rightmost Heavier described by the flowchart in Figure 2.22 runs in polynomial time in n, the length of the SLP which is input.*

Proof. There are only a few differences between this algorithm and Find Leftmost Heavier. We do not run the algorithm Get Length, we initialize t to 0 rather than to $k+1$, we output -1 rather than 1 as the second value to Find the Occurrence, and we do not check at the end to see if $h = -1$ or change t to -1. Not calling Get Length and not checking to see if $h = -1$ or changing t to -1 decrease the number of steps it takes to run Find Rightmost Heavier from the number of steps required to run Find Leftmost Heavier. The other differences do not affect the time required to run the program. Therefore, since Find Leftmost Heavier runs in polynomial time in n by Lemma 2.41, so does Find Rightmost Heavier. □

Lemma 2.44. *The algorithm Find First Occurrence Order described by the flowchart in Figure 2.23 inputs an SLP $\mathbb{B}$ and returns a list of indices of generators appearing in w_B which is ordered by the position of the leftmost occurrence of each generator or its inverse from smallest (leftmost) position to largest (rightmost).*

Proof. We begin by inputting an SLP $\mathbb{B}$, creating an empty list **fOL**, and initializing the variables *bigO*, short for 'biggest occurrence', and *bigOg*, short for 'biggest occurrence generator', to -1. Each item **fOL**$[i]$ in **fOL** will contain the position of the leftmost occurrence in w_B of the i^{th} generator or its inverse; the purpose of the first loop is to accomplish this.

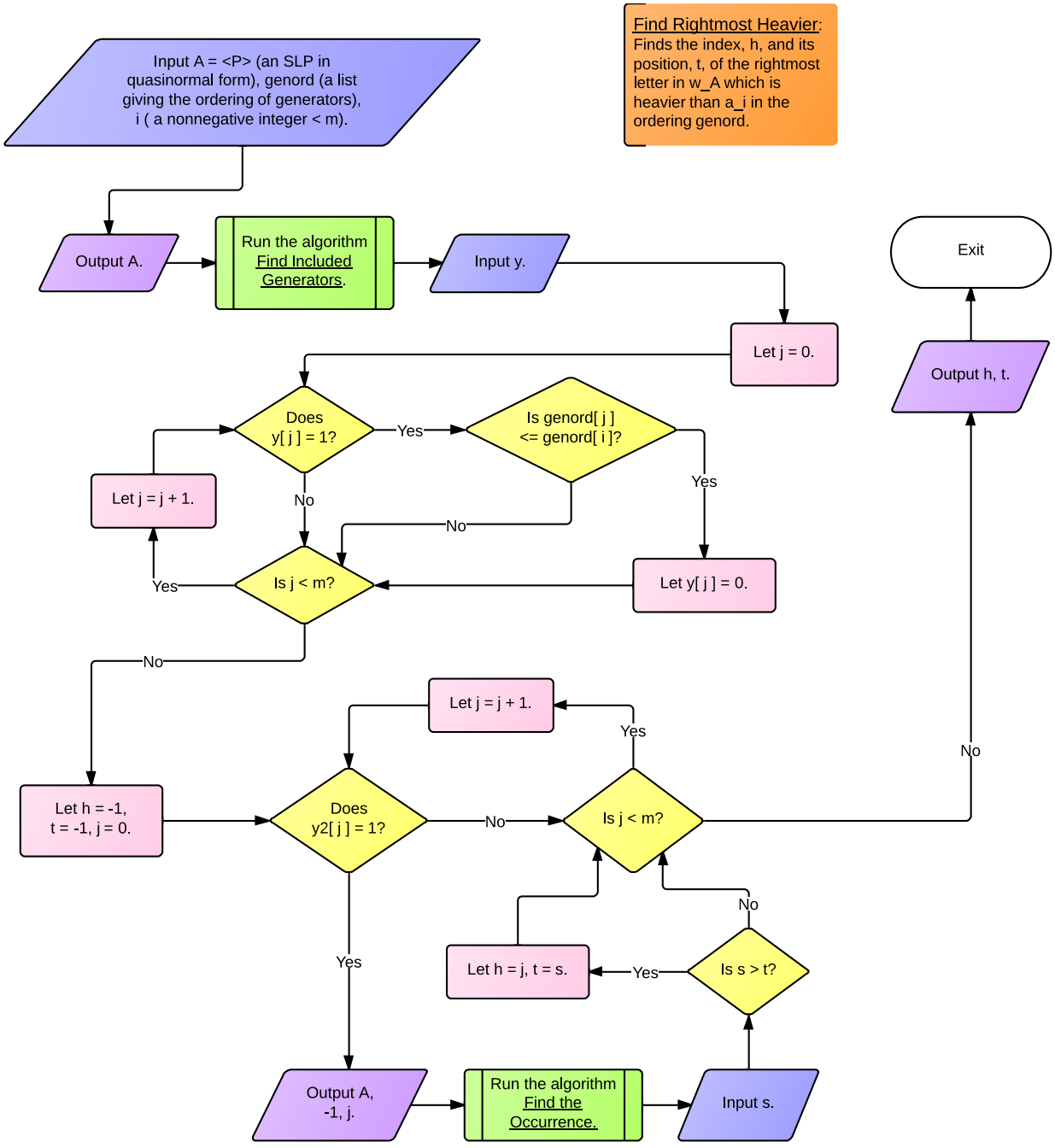


Figure 2.22: Algorithm Find Rightmost Heavier

We use i to step through the generators one at a time, starting with $i = 0$. We output $\mathbb{B}$, 1, and i to the algorithm Find the Occurrence, and by Lemma 2.28, the number indicating the position of the first, or leftmost, occurrence of $a_i^{\pm 1}$ in w_B is input. We assign this position to the variable O. If $O > bigO$, then we set $bigO$ to the value of O and i to $bigOg$. We then return to the beginning of the loop. Therefore at the end of the first loop, **fOL**$[i]$ is the position of the leftmost occurrence of $a_i^{\pm 1}$ for every $i = 0, 1, \ldots m-1$; $bigO$ is the largest of all the values in **fOL**, and $bigOg$ is the index in **fOL** of that value.

Before beginning the main loop, we create an empty list **fOoL** (short for 'first occurrence order list') and a list **didG** of length m with each item set to 0. **fOoL** is the list we will return at the end of the routine, and **didG** will allow us to keep track of which generators have already been added to **fOoL** or which have been discovered to not appear in w_B. For the main loop, we again use i to step through the generators, starting with $i = 0$. We begin each iteration of the main loop by letting $Lo = bigO$ and $Log = bigOg$. We then enter a smaller, nested loop in which we use j to step through the generators, starting with $j = 0$. This smaller loop is used to find the smallest position **fOL**$[j]$ for which j is not already in **fOoL**: In each iteration, if **fOL**$[j] < Lo$ and **didG**$[j] = 0$, then we let $Lo =$ **fOL**$[j]$ and $Log = j$. Then we return to the beginning of the smaller loop. Thus after stepping through the smaller loop m times, if $Lo > -1$, then Log is the index of the generator whose leftmost occurrence in w_B is left of the leftmost occurrence of all other generators whose indices are not yet in **fOoL**. In this case, we append Log to **fOoL**. If $Lo = -1$, then Log is the index of a generator not appearing in w_B. Whatever the value of Lo, we set **DidG**$[Log]$ to 1, so that we know that a_{Log} has been taken care of in the main loop, and return to the beginning of the main loop. Now when we exit the main loop, the item **fOoL**[0] will be the index of the generator whose leftmost occurrence in w_B is farthest left (that is, which occurs as the leftmost letter of w_B); **fOoL**[1] will be the index of the generator whose leftmost occurrence in w_B is farther left than any other generator except $a_{\mathbf{fOoL}[0]}$; and so on. We output **fOoL**. □

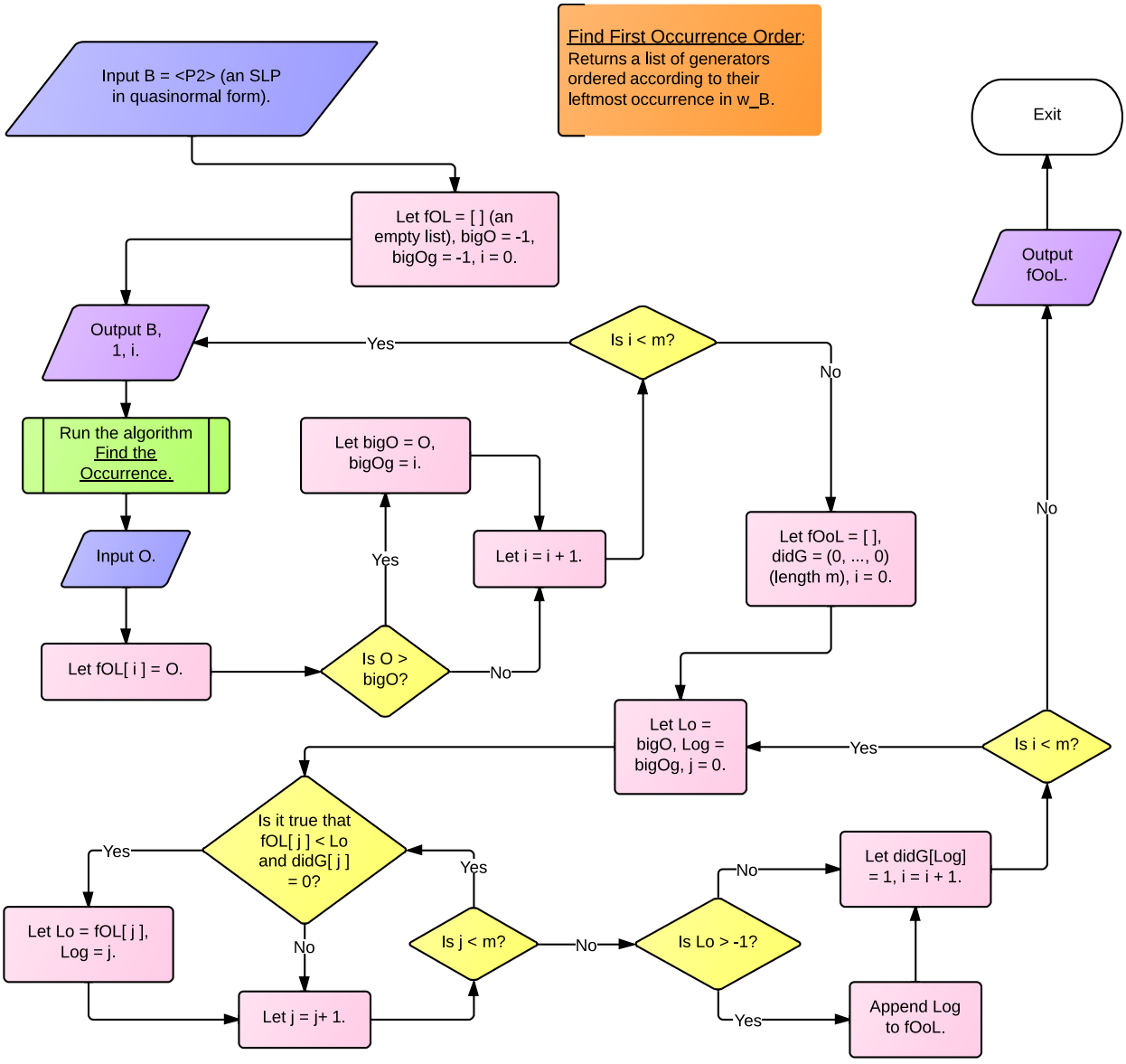


Figure 2.23: Algorithm Find First Occurrence Order

Lemma 2.45. *The algorithm Find First Occurrence Order described by the flowchart in Figure 2.23 runs in polynomial time in n, the length of the SLP which is input.*

Proof. The only part of Find First Occurrence Order which depends on n in any way is the call to the algorithm Find the Occurrence. This routine is called once for each iteration of the first loop, so m times total in Find First Occurrence Order. By Lemma 2.29, Find the Occurrence runs in polynomial time, say $p(n)$ steps. Thus the number of steps required to run Find First Occurrence Order is $mp(n) + c$, where c is the constant bound on the number of steps outside of the call to Find the Occurrence. Since $mp(n) + c$ is a polynomial in n, Find First Occurrence Order runs in polynomial time. □

2.4 Algorithms, Part 3 – Lexicographic Ordering

The algorithm we discuss next, Put In Lexicographic Order 2, is at the heart of putting a given word into lexicographic order. It takes two SLPs, $\mathbb{A}$ and $\mathbb{B}$, which produce words in lexicographic order according to a list, which is also input, and returns the SLP which produces the word which is similar to $w_A \cdot w_B$ and in lexicographic order according to that list. It is a very complex algorithm which would not fit on a single one-page flowchart, so we broke the flowchart into several pieces. All of the pieces which are not separate algorithms in the Python routines, but just part of the Put In Lexicographic Order 2 routine, are discussed within the proofs regarding Put In Lexicographic Order; only the flowcharts are separate. We will indicate which flowchart accompanies a certain piece when we begin to address that piece.

The list **genord** is used in Put In Lexicographic Order 2 as in some of the algorithms above, with each item **genord**$[j]$ set to the position of a_j in the ordering. For example, if **genord** is the list $[1, 3, 0, 2]$, this means that the generators a_0, a_1, a_2, and a_3 have been given the ordering a_2, a_0, a_3, a_1: The generator a_2 is the lightest, a_0 is second lightest; and so on.

Before delving into the proof of what the lemma does and how much time it requires, we give an overview of what the algorithm Put In Lexicographic Order 2 does. The algorithm acts upon the production rules, rather than the produced words, but in this overview we will consider what happens to the words and individual letters in them. Let us call the first word w_A and the second w_B. There is one large outer loop that may need to be repeated several times.

For each step of that loop, using the algorithm Find First Occurrence Order discussed above, we find the order in which the first occurrences of the generators appear in w_B. We loop through these generators, starting with the leftmost, moving to the second leftmost, and so forth. This is a smaller, nested loop that we will call the main loop. In this main loop we determine whether or not the leftmost occurrence of the current generator can and should move, possibly as the leftmost letter of a block of letters, to a particular position in w_A, and if so, move each with its block to where it must land in w_A in order for $w_A \cdot w_B$ to be in lexicographic order. Following is a list of steps that may occur for each iteration of the loop (not all of the steps occur in every iteration, and some or all may be repeated). We will discuss the steps further in the proof of Lemma 2.46. It may help to refer to Figure 2.24.

1. Let the leftmost occurrence of the generator for the current step be c.

2. Create a list of all generators in w_B which occur left of c and call the list **LofcGens**. If c does not commute with every generator in **LofcGens** then c cannot move now; in this case begin the loop again with the next generator.

3. Find the rightmost letter in w_A which does not commute with c and call it b. Then c cannot move left of b.

4. Find the leftmost letter in w_A which is right of b and heavier than c; call it z. Now c should move to the immediate left of z.

5. Find all generators which occur in w_A to the right of z and call this list **zRofzGens**.

6. Create a list **genlist** which is the union of **LofcGens** and **zRofzGens**.

7. Find the leftmost letter in w_B which is right of c and which fails to commute with at least one generator in **genlist**. Let this letter be h. The rightmost letter of the block to move must lie to the left of h in w_B.

8. Find the leftmost letter between c and h which is heavier than z; call it e. Let g be the letter to the immediate left of e. Now g is the rightmost letter in w_B which could be the rightmost letter of the block to move. However, there may be letters between c and g which should move farther left than c moves; we do not want to include these in our block.

9. Find all generators which occur to the right of b in w_A; call this list **RofbG**.

10. Find the leftmost letter between c and g, including g but not c, which commutes with b; let this letter be f. It is possible that f should move farther left than c does.

11. Find all generators occurring between c and f, including c but not f; let this list be **cf1Gens**.

12. Create a list **chkGList** which contains b, all of the generators in **RofbG**, and all of the generators in **cf1Gens**. If f moves farther left than c does, f must commute with all of the letters in **cf1Gens**.

13. Find the leftmost letter between f and g, including f and g, which commutes with all of the generators in **chkGList** and call this letter $newf$. If no such $newf$ exists, skip ahead to step 17. If $newf \neq f$, let $f = newf$ and return to step 11. If $newf = f$, proceed to the next step.

14. Find the rightmost letter left of b, or b itself, which is heavier than f. Call this letter d.

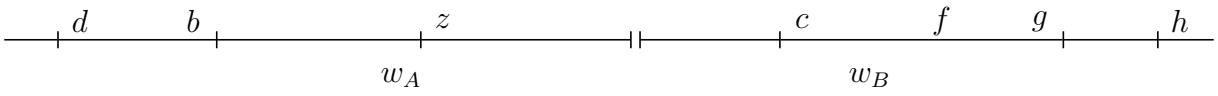


Figure 2.24: Putting $w_A \cdot w_B$ into Lexicographic Order

If f moves farther left than c moves, then f must move left of d.

15. Find all generators occurring between d and b, including d and b, and call this list **dbGens**. If f commutes with every generator in **dbGens**, then reassign g to be the letter just to the left of f and proceed to step 17. Otherwise, f moves with c, but a letter to the right of f in the cg-block may not move with c, so proceed to step 16.

16. Find the leftmost letter right of f in the cg-block which commutes with all of the letters in **chkGList**, reassign the variable f to this letter, and return to step 11.

17. Move the block of letters beginning with c and ending with g to the immediate left of z, and call the new first word now containing this block w_A and the new second word no longer containing this block w_B.

18. Get the next generator in the list, if there is one, and return to step 1.

Note that in the routine, c, b, z, and so forth are the indices of the specified generators, rather than the letters themselves; we used c, b, z, and so forth for the letters in the explanation above for the sake of readability. It remains to be seen that this process results in a new word which is similar to $w_A \cdot w_B$ and in lexicographic order and that this process runs in polynomial time in the sum of the sizes of $\mathbb{A}$ and $\mathbb{B}$.

Lemma 2.46. *The algorithm Put In Lexicographic Order 2 described by the flowchart in Figure 2.25 inputs two SLPs, $\mathbb{A}$ and $\mathbb{B}$, both in lexicographic order according to the list* **genord**, *which is also input, and it returns an SLP in quasinormal form which produces the word which is similar to $w_A \cdot w_B$ and in the lexicographic order defined by* **genord**.

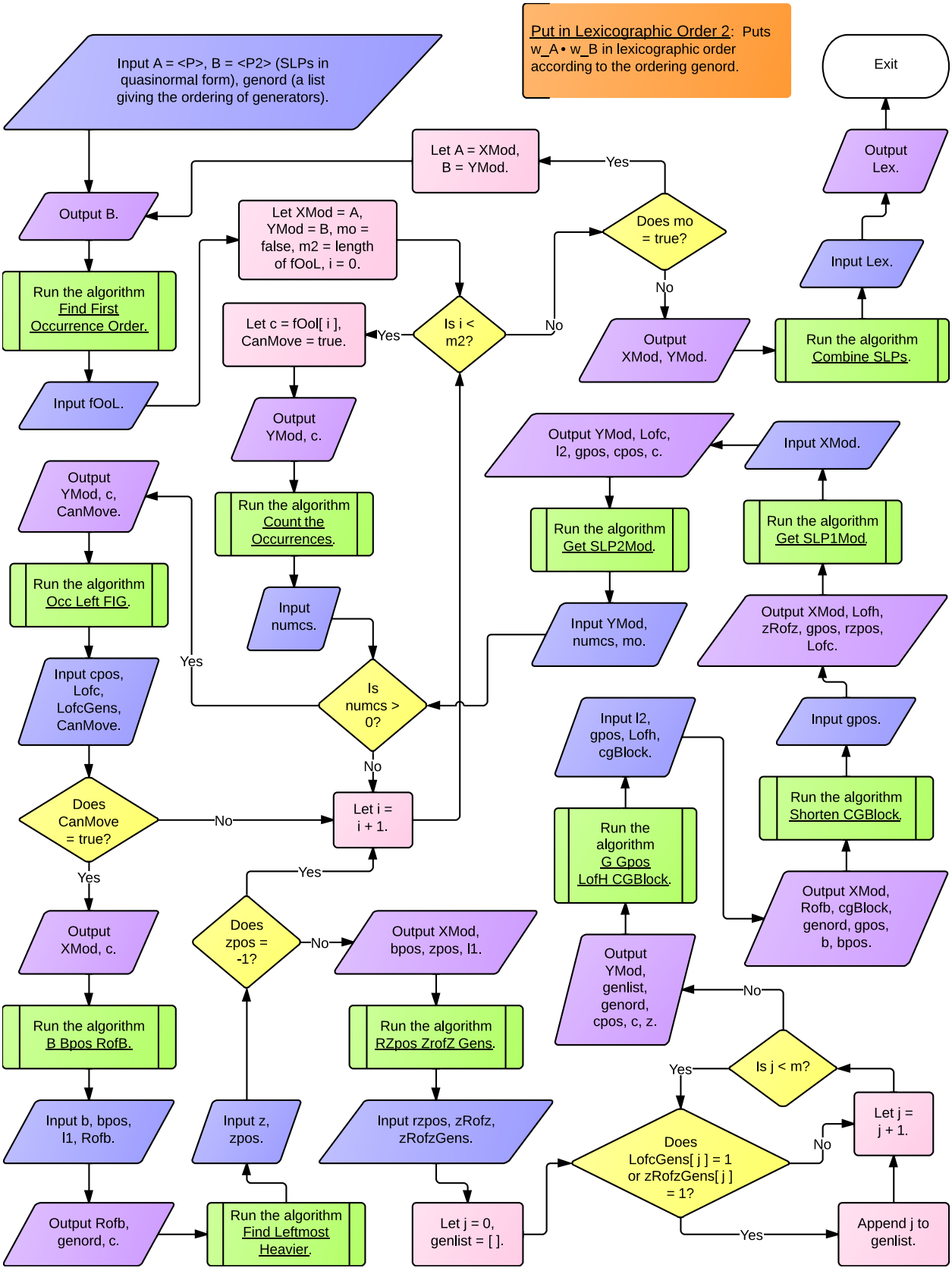


Figure 2.25: Algorithm Put In Lexicographic Order 2

Proof. After inputting SLPs $\mathbb{A}$ and $\mathbb{B}$ and the list **genord** giving the lexicographic ordering, we begin by initializing the variable tf to true. We will set tf to false when no more blocks of letters in w_B will move into w_A, ending the large outer loop. The large outer loop is not indexed by a counter; it merely repeats until tf is set to false. (The variable tf does not appear in the flowchart because of lack of space, but the flowchart still indicates the way in which tf controls the flow of the routine.)

After setting tf to true, we immediately enter the large outer loop. We begin each iteration of the outer loop by outputting $\mathbb{B}$ to the algorithm Find First Occurrence Order, which by Lemma 2.44 returns a list **fOoL** of generators ordered by their leftmost occurrence in w_B. We then initialize several variables, letting $\mathbb{X}\text{Mod} = \mathbb{A}$, $\mathbb{Y}\text{Mod} = \mathbb{B}$, $mo = \text{false}$, and $m2$ be the length of the list **fOoL**. For simplicity, we will use w_X and w_Y to indicate the words produced by $\mathbb{X}$Mod and $\mathbb{Y}$Mod, respectively. $\mathbb{X}$Mod and $\mathbb{Y}$Mod are copies of $\mathbb{A}$ and $\mathbb{B}$ which we will use and modify throughout the routine, *mo*, short for 'move occurred', gets set to true if any letters get moved from w_Y to w_X at any point during the current iteration of the large outer loop, and $m2$ is the number of iterations of the main loop which are performed during one iteration of the outer loop.

In the main loop we step through the generators in **fOoL**, in the order they occur in **fOoL**, using the index i, which runs from 0 to $m2 - 1$. Because of this, the generator for step i is not a_i as in many of the other algorithms; rather, the generator for step i is the generator whose index is **fOoL**$[i]$. We set $c =$ **fOoL**$[i]$, so a_c is the generator for step i; this is step 1 in the overview above. We next output $\mathbb{Y}$Mod and c to Count the Occurrences and input *numcs*, which by Lemma 2.18 is the number of occurrences of $a_c^{\pm 1}$ in w_Y. Here we enter a smaller loop, call it the *numcs* loop, inside the main loop. The *numcs* loop is repeated until $numcs = 0$. We know that we will enter the *numcs* loop at this point, because the list **fOoL** only contains the indices of those generators which occur in $\mathbb{Y}$Mod, so *numcs* must be at least 1.

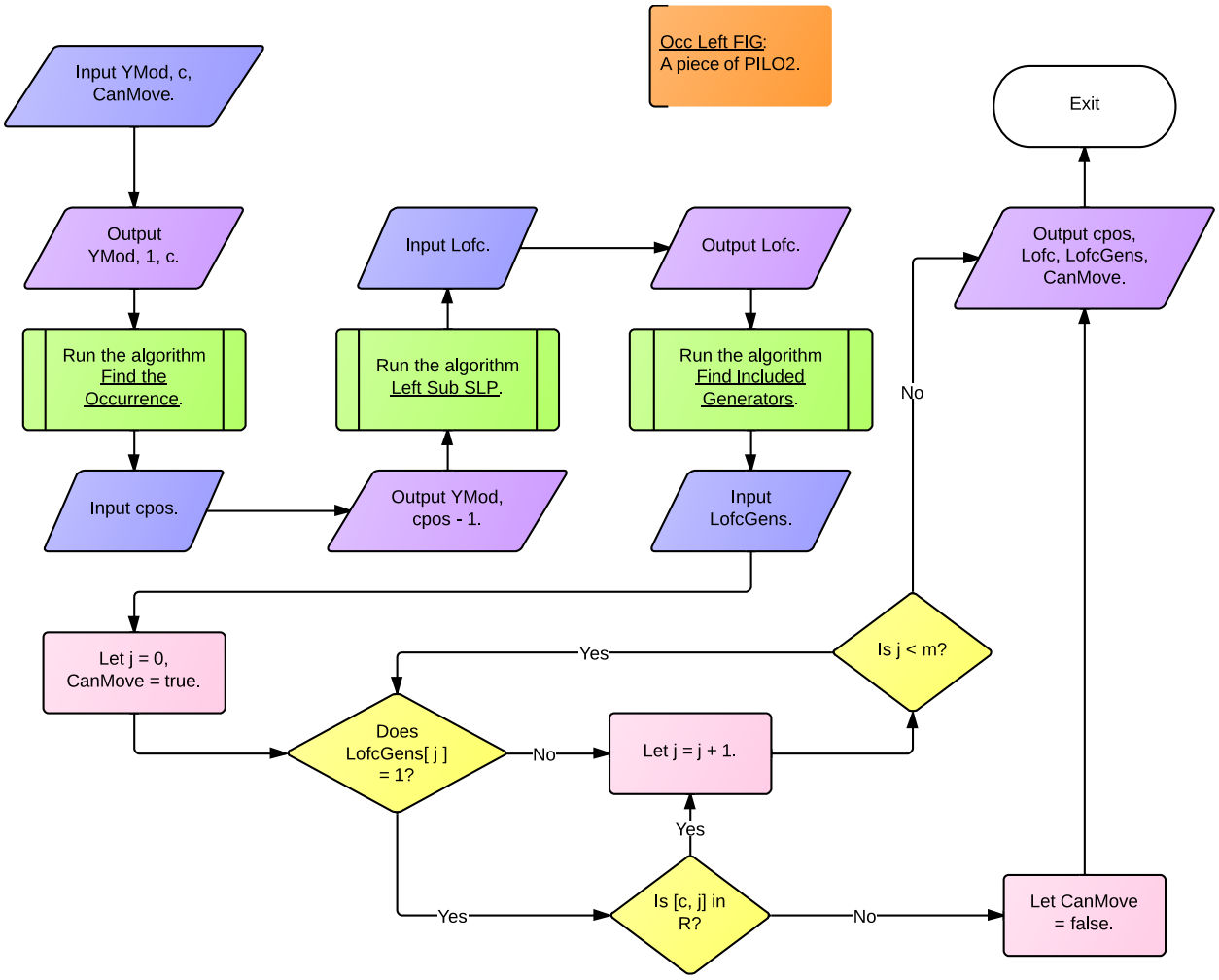


Figure 2.26: A piece of Put In Lexicographic Order 2

As we enter the *numcs* loop, we enter the part of the routine illustrated in the flowchart in Figure 2.26. We want to determine whether or not a_c commutes with all of the generators to its left in w_Y; if not, then the leftmost occurrence of a_c cannot move into w_X. To this end, we begin each iteration of the *numcs* loop by outputting 1 and c to the algorithm Find the Occurrence and then inputting *pos*. By Lemma 2.28, *cpos* is the position of the first occurrence of $a_c^{\pm 1}$ in w_Y. Next we output $\mathbb{Y}$Mod and $cpos - 1$ to Left Sub SLP, which by Lemma 2.22 returns an SLP $\mathbb{L}$ofc which produces the leftmost subword of $\mathbb{Y}$Mod having length $cpos - 1$. We then output $\mathbb{L}$ofc to Find Included Generators. This algorithm, by Lemma 2.6, returns a list **LofcGens** which indicates which generators occur in the word produced by $\mathbb{L}$ofc. Thus **LofcGens** indicates those generators which occur to the left of a_c in w_Y; we have finished step 2 in the list above.

Consider the first time we begin the *numcs* loop for the current c. Once we have the list **LofcGens** of generators occurring left of a_c in w_Y, we know that we have already applied the main loop to every generator in **LofcGens**, because of the order in which we are applying this loop to the generators. Therefore none of the generators in **LofcGens** will move left into w_X. Thus, in order for a_c to be able to move into w_X, a_c must commute with every generator in **LofcGens**; if not, we are finished with the main loop for this c. However, after the first time through the loop for the current c, there may be some letter in **LofcGens** to which we have not already applied the main loop, and which does not commute with a_c. In this case we move on to the next generator. However, it is possible that when we return to the larger outermost loop and begin going through the generators again, that some occurrence of the letter a_c in w_Y may move into w_X. This is why we must repeat the outer loop until no letters move; that is the only way we know we are finished. This may cause concern about the efficiency of the algorithm, but we will prove later that there is a polynomial limit on the number of blocks that will move.

To determine whether or not a_c commutes with every generator indicated in **LofcGens**, we

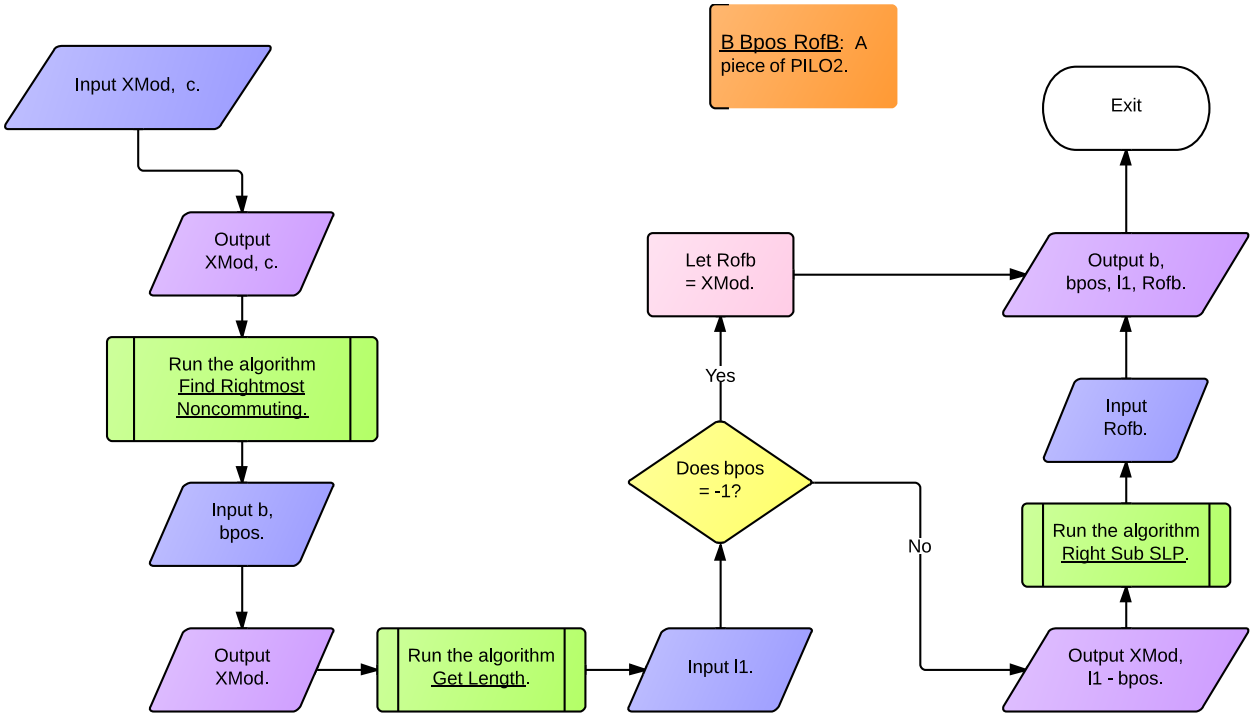


Figure 2.27: A piece of Put In Lexicographic Order 2

enter a small loop indexed by j in which we step through all of the generators, letting j run from 0 to $m-1$. For each iteration of this small loop, we check to see if **LofcGens**$[j] = 1$ (that is, if a_j is indicated in **LofcGens**); if not, we return to the beginning of the loop to check the next generator. If **LofcGens**$[j] = 1$, then we determine if either the pair (c, j) or the pair (j, c) is in **R**; if so, we return to the beginning of the small loop. If neither pair is in **R**, this means that there is some letter occurring to the left of a_c in w_Y with which a_c does not commute. Thus no occurrence of a_c can move left into w_X during this iteration of the large outermost loop. In this case, we set $CanMove$ to false and exit both the small loop and the *numcs* loop to return to the beginning of the main loop. This is the end of the part of the algorithm illustrated in Figure 2.26.

If a_c does commute with every generator in **LofcGens**, the next step (step 3 above) is to find the rightmost generator in w_X which blocks a_c; we will call this generator a_b. Then we know that the farthest left a_c can move in w_X is just to the right of a_b. This part of the algorithm

is shown in Figure 2.27. In order to find a_b, we output $\mathbb{X}$Mod and c to Find Rightmost Noncommuting, which by Lemma 2.34 returns the index b and position $bpos$ of the rightmost generator in w_X which does not commute with a_c, where $b = bpos = -1$ if no such generator exists. Thus if $b \neq -1$, we know it is possible for a_c to move just to the right of a_b, but not left of a_b; if $b = -1$, a_c can move all the way to the left in w_X. However, we do not know if a_c should move all the way left to land next to a_b; we just know that a_c should move left of any letter which is heavier than it (and lies to the right of a_b).

Thus in step 4 we find the leftmost letter in w_X which occurs to the right of a_b and which is heavier than a_c; we will call this letter a_z. The first step of finding this letter is to create an SLP $\mathbb{R}$ofb which produces the subword of w_X consisting of all of the letters to the right of a_b. Hence we first output $\mathbb{X}$Mod to Get Length and input, by Lemma 2.4, the length $l1$ of w_X. If $b = -1$, we let $\mathbb{R}$ofb = $\mathbb{X}$Mod. Otherwise, we output $\mathbb{X}$Mod and $l1 - bpos$ to the routine Right Sub SLP, which returns an SLP $\mathbb{R}$ofb whose produced word is the rightmost subword of w_X of length $l1 - bpos$, by Lemma 2.25. Thus this subword begins with the letter just to the right of a_b in w_X and ends with the last letter of w_X. This is the end of the part of Put In Lexicographic Order 2 illustrated in Figure 2.27. In order to find the leftmost letter which lies to the right of a_b and is heavier than a_c, we output $\mathbb{R}$ofb and c to Find Leftmost Heavier. By Lemma 2.40, this algorithm returns the index z and position $zpos$ of the leftmost letter in the word produced by $\mathbb{R}$ofb which is heavier than a_c, where $z = zpos = -1$ if no such letter exists. If $z = -1$, then even though a_c is able to, a_c should not move left into w_X, so we exit the *numcs* loop and return to the beginning of the main loop; otherwise we proceed.

Now we know that for the word produced by the algorithm to be in lexicographic order, we need for a_c to move just left of a_z. The key to making this algorithm sufficiently efficient, however, is to recognize that there may be a block of letters in w_Y beginning with a_c which should end up moving as a block to land just to the left of a_z; finding that block; and moving the whole block at one time. Steps 5 through 15 all involve finding that block, and in step 17

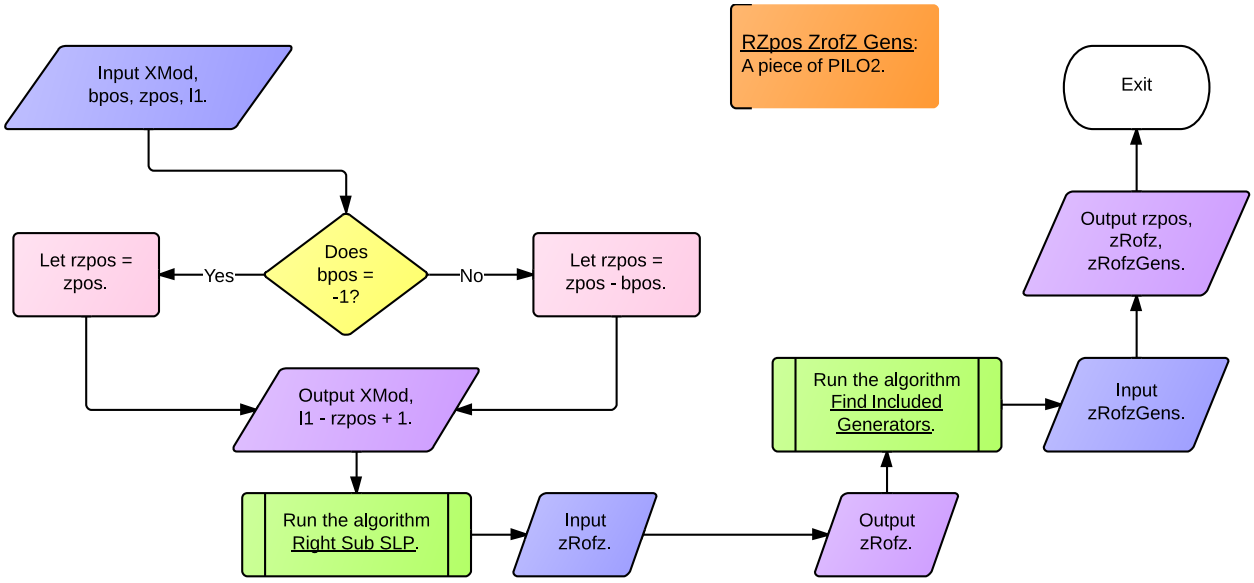


Figure 2.28: A piece of Put In Lexicographic Order 2

we move the block.

Figure 2.28 illustrates the part of the algorithm we discuss now. We know that every letter in the block that gets moved must commute with a_z and all of the letters between a_z and a_c. We already have a list, **LofcGens**, of generators in w_Y occurring to the left of a_c, but we need a list containing a_z and all of the generators in w_X occurring to the right of a_z. We find that list, **zRofzGens**, in step 5 above. To do this, we must first calculate the position $rzpos$ of a_z in w_X; $zpos$ is the position of a_z in the subword of w_X produced by $\mathbb{R}$ofb. If $bpos = -1$, we let $rzpos = zpos$; otherwise, we let $rzpos = zpos - bpos$. We use this to calculate the length of the subword of w_X beginning with a_z and ending with the last letter of w_X: since $l1$ is the length of w_X, the length we want is $l1 - rzpos + 1$. So we output $\mathbb{X}$Mod and $l1 - rzpos + 1$ to the algorithm Right Sub SLP, which by Lemma 2.25 returns an SLP $\mathbb{Z}$Rofz which produces the rightmost subword of w_X of length $l1 - rzpos + 1$. Next we output this SLP, $\mathbb{Z}$Rofz, to Find Included Generators and, by Lemma 2.6, input a list **zRofzGens** indicating which generators occur in the subword produced by $\mathbb{Z}$Rofz. This is the end of the part shown in Figure 2.28.

In step 6 we combine **zRofzGens** and **LofcGens** to get a list called **genlist** which contains a_z and all of the letters between a_z and a_c. To do this, we create an empty list named **genlist**, and we run a little loop using the index j to step through the generators, appending j to **genlist** if and only if **zRofzGens**$[j] = 1$ or **LofcGens**$[j] = 1$. Thus after running the loop, **genlist** contains the indices of every generator which is indicated in **zRofzGens** or **LofcGens**, and no indices of generators indicated in neither list.

The next part of the algorithm is shown in Figure 2.29. Since every letter in the block which gets moved with a_c must commute with every generator in **genlist**, in step 7 we find the leftmost letter a_h which lies to the right of a_c and which fails to commute with at least one generator in **genlist**; then a_c and every letter between a_c and a_h commute with all of the letters in **genlist**, and neither a_h nor any letter to the right of a_h can be part of the block. In order to accomplish this, we run the algorithm Get Length on $\mathbb{Y}$Mod to get, by Lemma 2.4, the length $l2$ of w_Y. Now $l2 - cpos + 1$ is the length of the subword of w_Y beginning with a_c and ending with the last letter of w_Y, the subword we want to check for such an a_h. So we output $\mathbb{Y}$Mod and $l2 - cpos + 1$ to Right Sub and input $\mathbb{C}$Rofc, an SLP which produces the subword of a_c beginning with a_c and ending with the last letter of w_Y. Next we output $\mathbb{C}$Rofc and **genlist** to Find Leftmost Noncommuting - List. By Lemma 2.36, this returns the index h and position $hpos$ of the leftmost letter in the word produced by $\mathbb{C}$Rofc which fails to commute with at least one generator in **genlist**. At this point any letters that may move in the block with a_c lie between a_c and a_h, or to the right of a_c if no such a_h exits. Just as with a_c itself, however, a letter should not move to the immediate left of a_z unless it is lighter than a_z.

Therefore the next step, step 8, is finding the leftmost letter in the word produced by $\mathbb{C}$Rofc which lies left of a_h (if a_h exists) and is heavier than a_z, and then assigning the letter to its immediate left to a_g, and the position of a_g to $gpos$. The letter a_g is potentially the rightmost letter of the block which will move just to the left of a_z. Now by design of the algorithm,

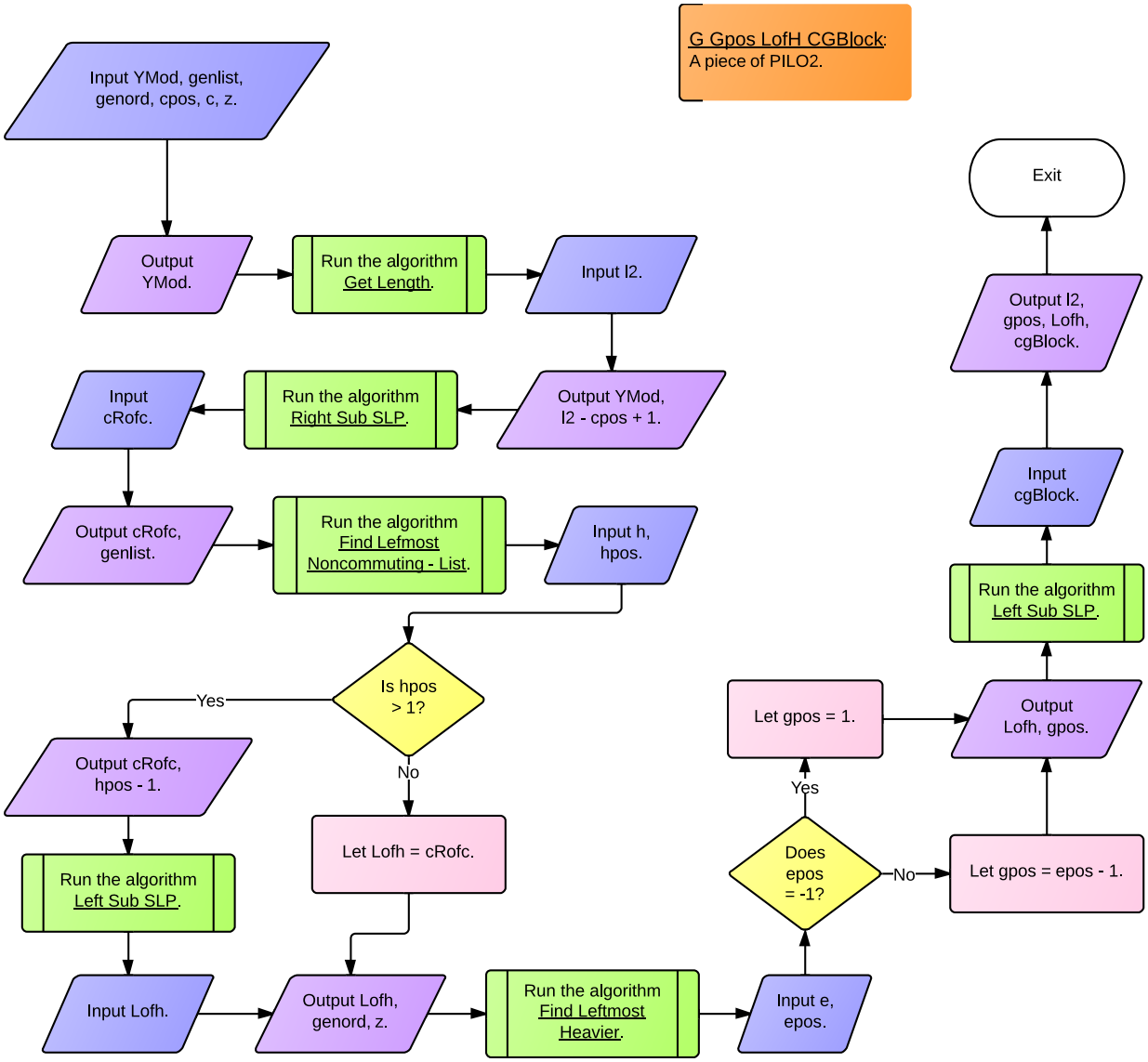


Figure 2.29: A piece of Put In Lexicographic Order 2

we know that a_c commutes with all of the generators in **genlist**, so *hpos* cannot be 1. If $hpos = -1$, then a_c and all letters to its right in w_Y commute with every generator in **genlist**, so we let $\mathbb{L}\text{ofh} = \mathbb{C}\text{Rofc}$. Otherwise $hpos > 1$, and in this case, we output $\mathbb{C}$Rofc and $hpos - 1$ to Left Sub SLP and input an SLP $\mathbb{L}$ofh which produces the subword of w_Y beginning with a_c and ending with the letter to the immediate left of a_h. After outputting $\mathbb{L}$ofh, *genord*, and z to Find Leftmost Heavier, by Lemma 2.40 we input the index e and position *epos* of the leftmost letter between a_c and a_h, including a_c, which is heavier than a_z. If $epos = -1$, we then set *gpos* to 1, since in this case a_c, the letter in position 1 of $\mathbb{L}$ofh, is the rightmost letter of the *cg*-block. Otherwise we set *gpos* to $epos - 1$. Although there is no need in the algorithm to indicate this, we think of the letter to the immediate left of a_e, or a_c itself if $epos = -1$, as a_g. Now we know that a_c, a_g, and all of the letters between them will at some point in the algorithm move left of a_z. However, some of the letters in the block beginning with a_c and ending with a_g, that is, the *cg*-block, may need to be moved left of where a_c is moved to. At first thought, this may seem impossible since w_Y is in lexicographic order. However, consider the following example.

Example 2.47. Let a_0, a_1, a_2, and a_3 be generators in lexicographic order; suppose the generators all commute with each other except for a_0 and a_2; and suppose $w_A = a_2a_3$ and $w_B = a_0a_1$. Let a_0 be the letter a_c in our discussion above. Then a_2 is a_b and a_3 is a_z; a_0 should move to be between a_2 and a_3. The letter a_1, however, commutes with all of the other letters and is lighter than a_2, so a_1 should move left of a_2; the word which is similar to $w_A \cdot w_B = a_2a_3a_0a_1$ and in lexicographic order is $a_1a_2a_0a_3$.

At this point the block we want to move begins with a_c and contains only letters that should move with a_c to land just left of a_z; we are trying to discover where that block ends. In our example above, we would not want to move a_1 with a_0; we would move a_1 later in the algorithm. Thus we want to find the leftmost letter a_f between a_c and a_g, including a_g, which should end up moving farther left in w_X than a_c moves. First we need an SLP which

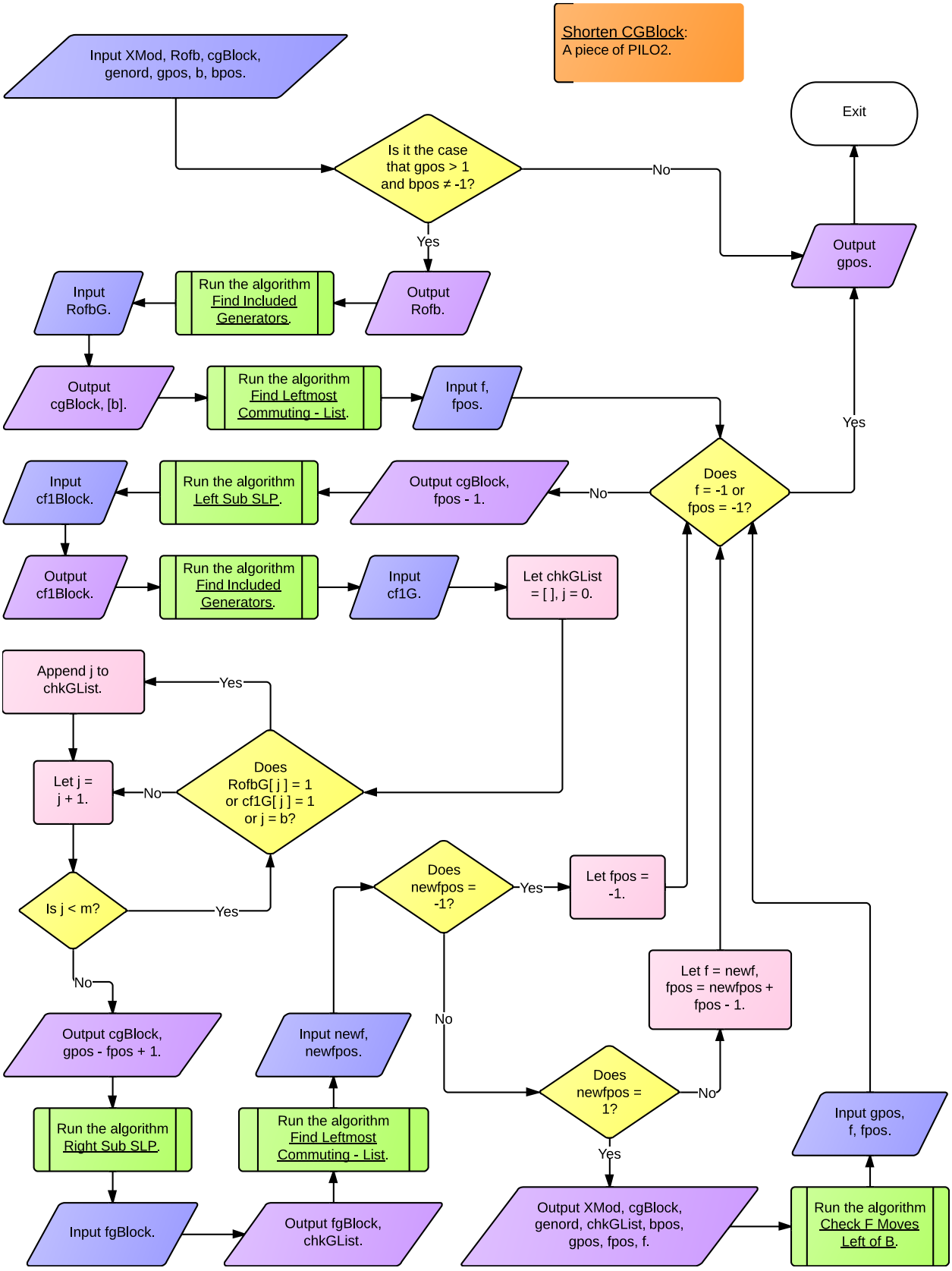


Figure 2.30: A piece of Put In Lexicographic Order 2

produces the *cg*-block, that is, the subword of w_Y beginning with a_c and ending with a_g. Hence we output $\mathbb{L}$ofh and *gpos* to Left Sub SLP and then input an SLP $\mathbb{CG}$Block which produces the *cg*-block by Lemma 2.22. This ends the part of the routine illustrated in 2.29; the next part is shown in 2.30.

Now all of the letters between a_b and a_z are lighter than a_c, because a_z was chosen to be the leftmost letter right of a_b which is heavier than a_c. Furthermore, any letter in the *cg*-block which should move farther left than a_c moves must be heavier than a_c since it commutes with a_c but lies to the right of a_c in the lexicographically ordered word w_Y. Thus any such letter is heavier than a_c and so is also heavier than all of the letters between a_b and where a_c will land. However, it should not move farther left than a_c moves unless it is lighter than some letter left of a_z to the left of which it may move. Therefore any letter which should move left of a_c must end up moving left of a_b, and so must commute with a_b and every letter to the right of a_b in w_X. Thus in step 9, we create a list named **RofbG** containing all of the generators which occur to the right of a_b in w_X, which we will use in step 12. This is done by outputting $\mathbb{R}$ofb to Find Included Generators, which returns a list **RofbG** indicating which generators occur to the right of a_b in w_X.

Rather than first finding the leftmost letter a_f in the *cg*-block which commutes with all of the letters to the right of a_b in w_X, however, we first find the leftmost letter lying to the right of a_c in the *cg*-block which commutes with a_b; this is step 10. (Later we will combine **RofbG** with a couple of other lists of generators to see if the letter with which we are concerned at the time commutes with all of those generators.) To find such an a_f, we output $\mathbb{CG}$Block and a list containing the single item b to Find Leftmost Commuting - List. By Lemma 2.38 we input the index f and position $fpos$ of the leftmost letter in the *cg*-block which commutes with a_b. Now if no such a_f exists, that is, if $f = -1$ or $fpos = -1$, then there is no letter in the *cg*-block which will move farther left than a_c moves, and so we know now that a_g is the end of the block which will move with a_c. In this case, we skip ahead to where we move the

cg-block (step 17 in the list before this lemma); otherwise, we enter a smaller loop we will call the f loop (still inside the *numcs* loop) which repeats until $f = -1$ or $fpos = -1$; we will explain this as we continue discussing this algorithm.

We know that in order for a_f to need to move left of a_b, some letter left of a_b or a_b itself must be heavier than a_f, and a_f must be able to commute to land left of such a letter. If this is not the case, then we want a_f to be included in the block with a_c which gets moved in step 17, so we need to determine whether this is the case or not. We know that a_f commutes with a_z and all of the letters between a_z and a_c, but we must find out if a_f also commutes with a_c, the letters between a_c and a_f, and the letters between a_b and a_z. Hence in step 11 we create a list **cf1Gens** of all generators occurring in the *cg*-block to the left of a_f. We do so in the following way at the beginning of each iteration of the f loop. We first output $\mathbb{C}\mathbb{G}$Block and $fpos - 1$ to Left Sub SLP and input $\mathbb{C}$f1Block, an SLP producing the subword of the *cg*-block beginning with a_c and ending with the letter to the immediate left of a_f. Then we output $\mathbb{C}$f1Block to Find Included Generators to get a list **cf1G** indicating which generators occur in the subword produced by $\mathbb{C}$f1Block.

This still is not the entire list of generators with which a_f must commute if it is to move left of a_b, so in step 12 we create a list **chkGList** containing a_b, the generators in **RofbG**, and the generators in **cf1Gens**. To accomplish this, we first create an empty list called **chkGList** and then run through a small loop using j to step through the generators. In each iteration of the loop, we append j to **chkGList** if and only if **RofbG**$[j] = 1$ or **cf1Gens**$[j] = 1$ or $j = b$. Hence as we exit the loop, **chkGList** contains the indices of all the generators with which a_f must commute in order for it to be able to move left of a_b.

For the sake of efficiency, rather than just checking to see if f commutes with every generator in **chkGList**, we find the leftmost letter in the fg-block which commutes with every generator in **chkGList**. To do so we first output $\mathbb{C}\mathbb{G}$Block and $gpos - fpos + 1$ to Right Sub SLP to get the SLP $\mathbb{F}$gBlock which produces the subword of w_Y beginning with a_f and ending with

a_g, and then we output $\mathbb{F}$gBlock and **chkGList** to Find Leftmost Commuting - List to get the index $newf$ and position $newfpos$ of the leftmost letter in the fg-block which commutes with all of the generators in **chkGList**. Now if $newfpos = -1$, then there is no letter in the cg-block which moves left of where c will land, so we want to skip ahead to step 17. We do so by letting $fpos = -1$ in order to exit the f loop and move on to the next part of the algorithm. If $newf \neq -1$ then $newf$ is the leftmost letter in the cg-block which possibly moves left of b. If $newf = f$, then we continue with the rest of the steps in the f loop; otherwise we let $f = newf$ and $fpos = newfpos - fpos + 1$, the position of a_{newf} in the cg-block rather than the fg-block, and return to step 11; that is, we return to the beginning of the f loop.

If $newf = f$, we still need to check and see if there is a letter left of a_b, or a_b itself, which is heavier than a_f and to the left of which a_f can commute; if not, then a_f will move in the block with a_c. Thus in step 14, we find the rightmost letter left of a_b, or a_b itself, which is heavier than a_f. Although we have not yet finished the part of the routine illustrated in Figure 2.30, we enter a smaller part which is illustrated in Figure 2.31. In order to find such a letter, we first set $fMovesLofb$ to true and then output $\mathbb{X}$Mod and *bpos* to Left Sub SLP, which returns an SLP $\mathbb{B}$Lofb that produces the subword of w_X starting at the beginning of w_X and ending with a_b. Next we output $\mathbb{B}$Lofb, *genord*, and f to Find Rightmost Heavier, which by Lemma 2.42 returns the index d and position *dpos* of the rightmost generator in the word produced by $\mathbb{B}$Lofb which is heavier than a_f. If $dpos = -1$, then no such generator exists, so we set $fMovesLeftofb$ to false; otherwise, we continue with the steps in the following paragraph.

If such an a_d does exist, then we need to know if a_f can commute to be left of it. We know a_f can commute as far left as just to the left of a_b, but we need to know which generators are between a_d and a_b and whether or not a_f commutes with all of them and with a_d. Thus in step 15, we create a list **dbGens** of all the generators occurring in the db-block. To do

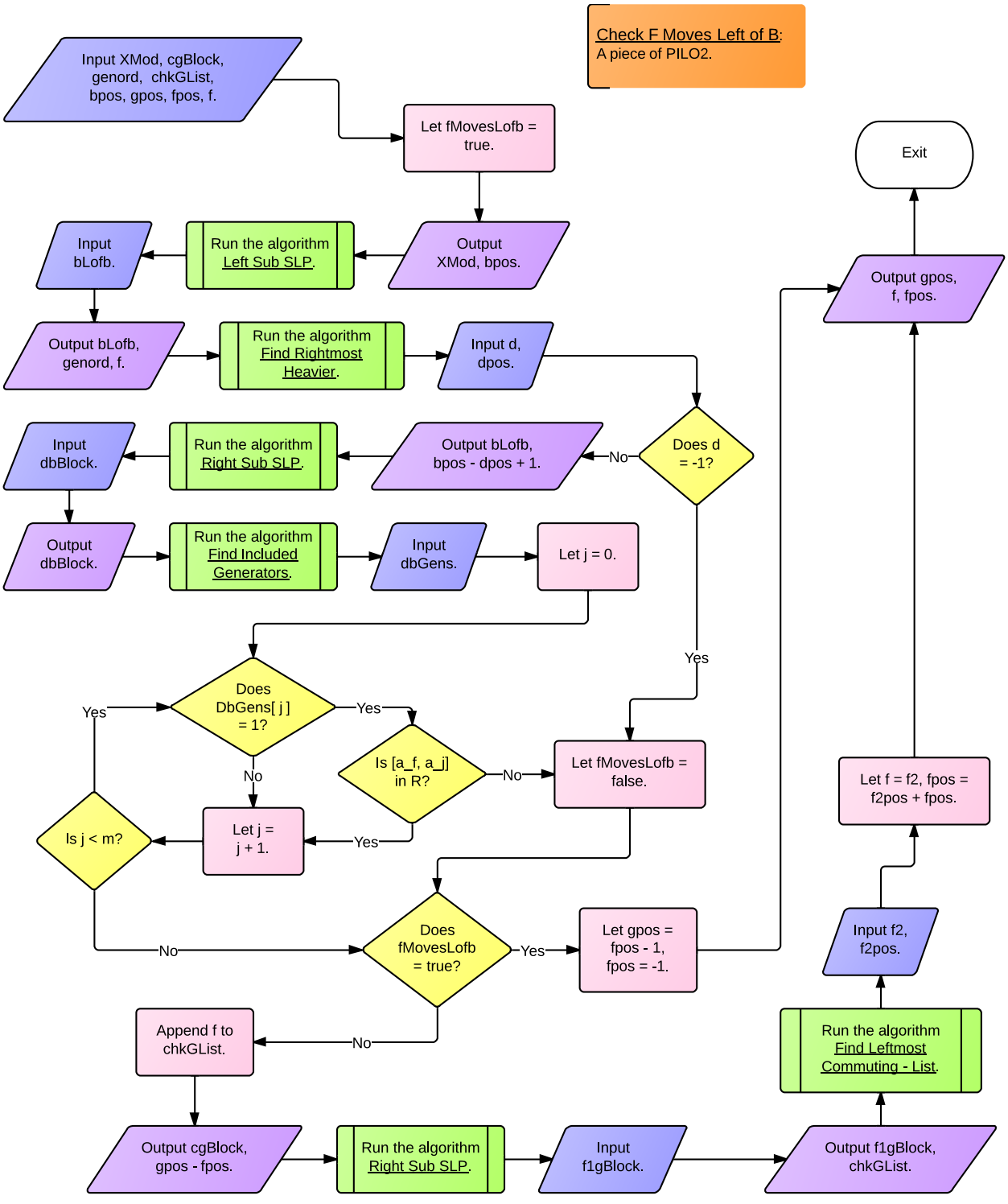


Figure 2.31: A piece of Put In Lexicographic Order 2

so, we first output $\mathbb{B}$Lofb and $bpos - dpos + 1$ to Right Sub SLP, which gives us an SLP $\mathbb{D}$bBlock; the word produced by $\mathbb{D}$bBlock is the block beginning with a_d and ending with a_b, that is, the *db*-block. Next we output $\mathbb{D}$bBlock to Find Included Generators to get the list **dbGens**, which indicates which generators occur in the *db*-block. We then enter a little loop (still inside the f loop) in which we use the index j to step through the generators, beginning with $j = 0$. In each iteration of the loop, we begin by checking whether or not **dbGens**$[j] = 1$; if not, we increase j by 1 and return to the beginning of the loop. If **dbGens**$[j] = 1$, then a_j occurs in the *db*-block, so we check to see if either of the pairs (a_j, a_f) and (a_f, a_j) is in **R**. If so, then a_f commutes with that letter a_j, so we return to the beginning of the loop to check the next generator. If neither (a_j, a_f) nor (a_f, a_j) is in **R**, we exit the loop and set $fMovesLeftofb$ to false. If we finish the little loop without setting $fMovesLeftofb$ to false, then a_f is able to commute just to the left of a_d.

If at this point in the routine $fMovesLeftofb$ is true then since a_f will move farther left than a_c, we do not want a_f to move in the block with a_c. We chose a_f to be the leftmost letter in the *cg*-block which might possibly move farther left than a_c, so we know that all of the letters between a_f and a_c will move with a_c to land just to the left of a_z; none of them should move farther. While we do not know the index of the generator lying just to the left of a_f, its position is $fpos - 1$, and we know that the subword of the *cg*-block beginning with a_c and with the letter to the immediate left of a_f is the block that will get moved in this iteration of the main loop. Thus we set the variable $gpos$ to $fpos - 1$ and the variable $fpos$ to -1. Letting $fpos = -1$ will allow us to break out of the f loop. Although we do not set g to the index of the generator just to the left of a_f (because we do not know it and do not use it during the remainder of the routine), let us call this letter a_g and the block beginning with a_c and ending with this letter the *cg*-block.

If, on the other hand, $fMovesLeftofb$ is false, then while we know that a_f will move with a_c, there may be a letter to the right of a_f in the *cg*-block which will move left of a_b. We

know that any other occurrence of the generator a_f will not move left of a_b, so we append f to **chkGList** and then in step 16 we find the leftmost letter right of a_f in the cg-block which commutes with all of the generators in **chkGList**. Recall that a generator is not considered able to commute with itself; we have no occurrence of (a_k, a_k) for any k in **R**. Thus the leftmost letter in the cg-block right of a_f which commutes with all of the generators in **chkGList** will not be any occurrence of the current a_f or any previous a_f (since those, too, will have been appended to **chkGList**). We first output $\mathbb{CG}$Block and $gpos - fpos$ to Right Sub SLP, which returns an SLP $\mathbb{F}$1gBlock that produces the subword of the cg-block beginning with the letter to the immediate right of a_f and ending with a_g. We output $\mathbb{F}$1gBlock and **chkGList** to Find Leftmost Commuting - List and input the index $f2$ and position $f2pos$ of the leftmost letter right of a_f which commutes with all of the generators in **chkGList**. We then reassign f by setting it to the value of $f2$, and we set $fpos$ to $f2pos + fpos$ so that $fpos$ is the position of this new a_f in the cg-block, rather than in the subword produced by $\mathbb{F}$1gBlock.

Whether $fMovesLeftofb$ was true or false, we now exit the part of the routine shown in Figure 2.31 and return to the part shown in Figure 2.30. We return to the beginning of the f loop, checking whether either $f = -1$ or $fpos = -1$; if either is true we exit the f loop; otherwise we proceed through it again. Once we exit the f loop we have finished with the part of the routine shown in Figure 2.30. At this point we know the position $gpos$ of the rightmost letter which will move in a block with a_c to land just to the left of a_z; it remains to move that block, which is step 17 in the list above.

When we say we move the block, what we mean is that we create an two SLPs: The first produces a concatenation of the subword of w_X beginning with the first letter of w_X and ending with the letter just to the left of a_z; the block to move; the subword of w_X beginning with a_z and ending with the last letter of w_X; and the subword of w_Y beginning with the leftmost letter and ending with the letter to the immediate left of a_c. The second produces the subword of w_Y beginning with the letter just to the right of a_g and ending with the last

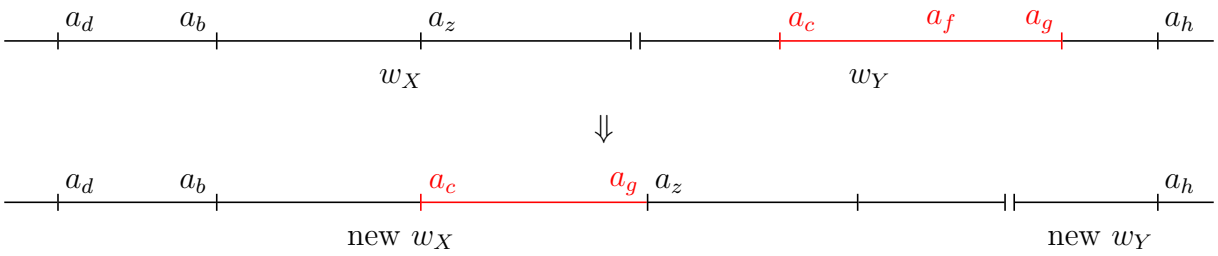


Figure 2.32: Moving the *cg*-Block

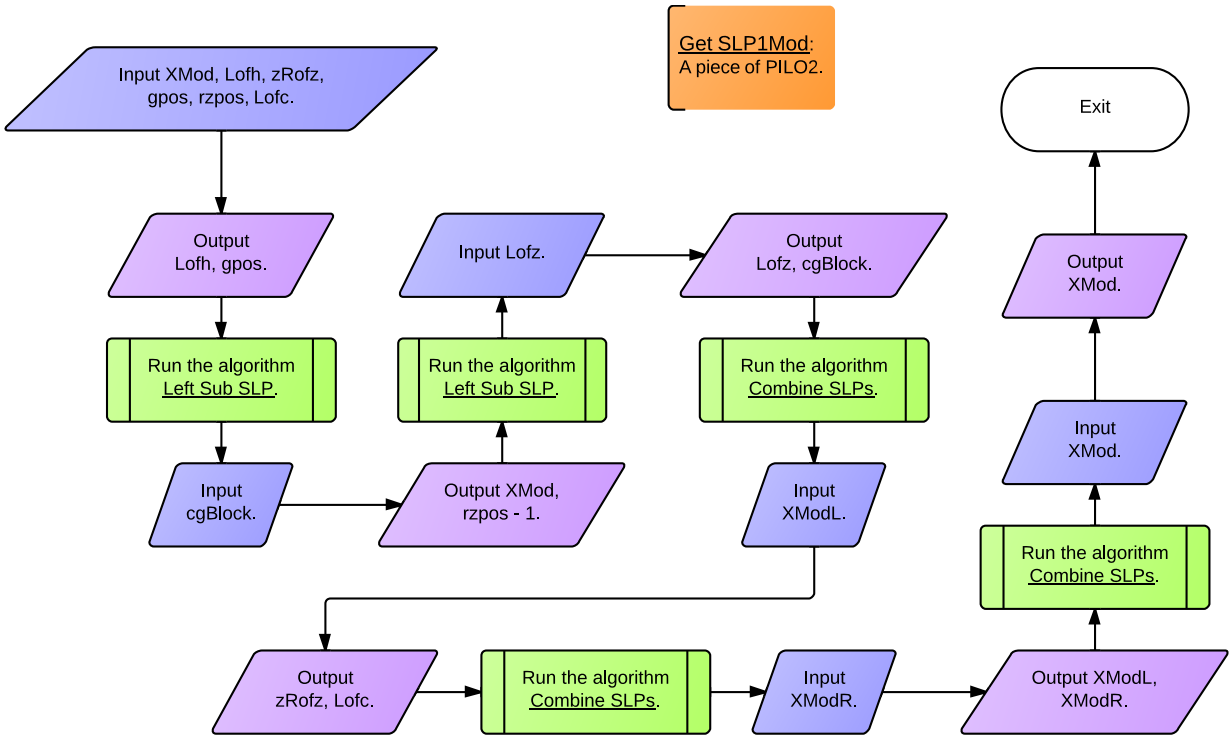


Figure 2.33: A piece of Put In Lexicographic Order 2

letter of w_Y. See Figure 2.32 for an illustration of this.

We now enter the part of the routine illustrated in Figure 2.33. To create the first SLP, we begin by outputting $\mathbb{L}$ofh and *gpos* to Left Sub SLP to get an SLP which produces the new *cg*-block. We reassign the variable $\mathbb{CG}$Block by setting it equal to this SLP. Next we output $\mathbb{X}$Mod and $rzpos - 1$ to Left Sub SLP and input as a result an SLP $\mathbb{L}$ofz which produces the subword of w_X beginning with the first letter of w_X and ending with the letter just to the left of a_z. Next we output $\mathbb{L}$ofz and $\mathbb{CG}$Block to the algorithm Combine SLPs. By Lemma 2.20,

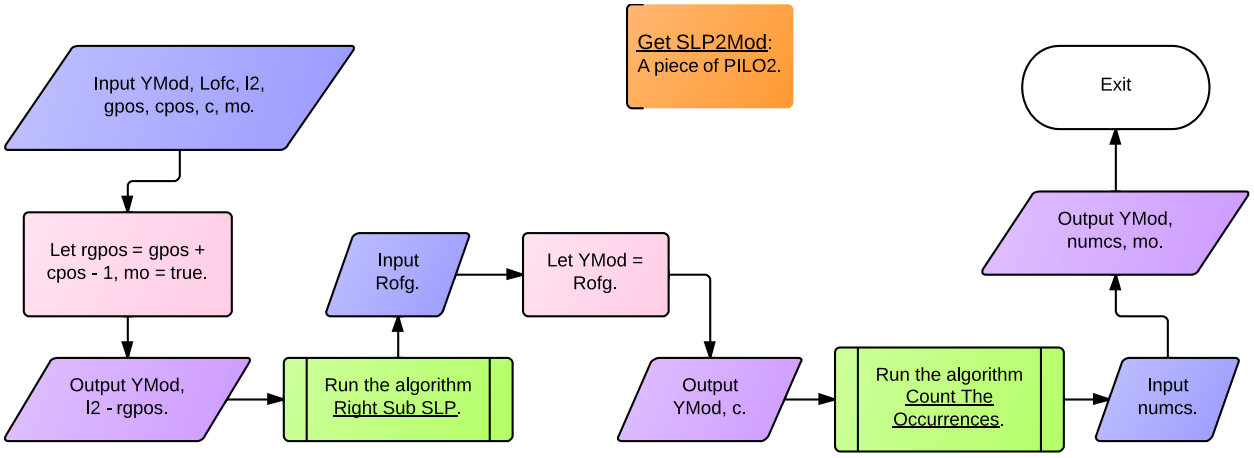


Figure 2.34: A piece of Put In Lexicographic Order 2

we input an SLP $\mathbb{X}$ModL which produces the concatenation of the word produced by $\mathbb{L}$ofz and the word produced by $\mathbb{C}\mathbb{G}$Block. We then output $\mathbb{Z}$Rofz and $\mathbb{L}$ofc to Combine SLPs and input an SLP $\mathbb{X}$ModR, which produces the concatenation of the words produced by $\mathbb{Z}$Rofz and $\mathbb{L}$ofc. Finally, we concatenate the words produced by $\mathbb{X}$ModL and $\mathbb{X}$ModR by outputting these two SLPs to Combine SLPs. We reassign $\mathbb{X}$Mod to the SLP which we input as a result of this last call to Combine SLPs. This is the end of the part of the routine shown in Figure 2.33.

We proceed to the part illustrated in Figure 2.34. We begin the process of forming the second new SLP by letting $rgpos = gpos + cpos - 1$ so that $rgpos$ is the position of a_g in w_Y rather than in the word produced by $\mathbb{L}$ofh. We also set mo to true, because a cg-block is moving in this iteration of the larger outermost loop. Next we output $\mathbb{Y}$Mod and $l2 - rgpos$ to Right Sub SLP, which returns an SLP $\mathbb{R}$ofg that produces the subword of w_Y beginning with the letter to the immediate right of a_g and ending with the last letter of w_Y. Finally, we set $\mathbb{Y}$Mod equal to this new SLP. Before returning to the beginning of the $numcs$ loop, we output $\mathbb{Y}$Mod and c to Count the Occurrences, which, by Lemma 2.18, returns the number of occurrences of a_c in the new w_Y; we set $numcs$ to this number. This is the end of the part shown in Figure 2.34.

Once *numcs* reaches 0 for this a_c, we exit the *numcs* loop and return to the beginning of the main loop in order to begin the process again with the next generator in list **fOoL**. After progressing through the main loop for every generator in **fOoL**, we check to see if $mo = true$. This is the case if and only if step 17 occurred for any generator. If $mo = true$, we let $\mathbb{A} = \mathbb{X}\text{Mod}$ and $\mathbb{B} = \mathbb{X}\text{Mod}$, and we return to the beginning of the main loop to the repeat the entire process for any generators occurring in the latest w_Y.

When $mo = false$, we exit the main loop. We want to create an SLP which produces the word $w_X \cdot w_Y$, where w_X and w_Y are now the words produced by the most recent assignment of $\mathbb{X}\text{Mod}$ and $\mathbb{Y}\text{Mod}$, respectively. Therefore we output $\mathbb{X}\text{Mod}$ and $\mathbb{Y}\text{Mod}$ to Combine SLPs and input an SLP $\mathbb{L}\text{ex}$ which is in quasinormal form and produces the word $w_X \cdot w_Y$. The SLP $\mathbb{L}\text{ex}$ is the SLP we output at the end of Put In Lexicographic Order 2. Let w_L be the word produced by $\mathbb{L}\text{ex}$.

The first time a *cg*-block is moved, the part of w_Y which lies to the left of the *cg*-block is removed from w_Y and concatenated to the end of w_X. Since w_Y was in lexicographic order before the block was moved, any subword of w_Y is in lexicographic order. Thus the new w_Y, which is a rightmost subword of the previous w_Y, is in lexicographic order. Thus every time a *cg*-block is moved, the new w_Y is in lexicographic order. Furthermore, by construction, each *cg*-block is moved to the place in w_X such that the concatenation of the words produced by $\mathbb{L}\text{ofz}$, $\mathbb{CG}\text{Block}$, and $\mathbb{Z}\text{Rofz}$ is in lexicographic order. Since the subword of w_Y produced by $\mathbb{L}\text{ofc}$ is in lexicographic order and contains no letters which should move into w_X, the word which results from concatenating the word produced by $\mathbb{L}\text{ofc}$ to the end of the concatenation of the words produced by $\mathbb{L}\text{ofz}$, $\mathbb{CG}\text{Block}$, and $\mathbb{Z}\text{Rofz}$ is in lexicographic order. In other words, by construction, for every letter $w_L[i]$ in w_L, all of the letters between $w_L[i]$ and the rightmost letter left of $w_L[i]$ with which $w_L[i]$ does not commute weigh less than $w_L[i]$ in the lexicographic ordering given by **genord**. Thus, by Lemma 1.2, the word produced by $\mathbb{L}$ is in lexicographic order. □

Before we can prove that the algorithm Put in Lexicographic Order 2 runs in polynomial time, we must prove several facts about the number of *cg*-blocks that move when Put in Lexicographic Order 2 is run. The first three of these together provide a bound for the number of subwords that move from the original w_B into the original w_A. These lemmas and definitions which we discuss before showing that Put in Lexicographic Order 2 runs in polynomial time are all included in the context of the algorithm Put in Lexicographic Order 2. Whenever we speak of where a subword is or lies, we are speaking of the position of the subword in the original w_A or w_B, before any subwords are moved. When a subword moves, we say it *lands* in its new position.

Lemma 2.48. *Suppose x and c are two distinct generators of a RAAG, with $x < c$. Assume that x_1 and x_2 are two occurrences of x in w_A, with x_1 left of x_2, and that c_1 and c_2 are two occurrences of c in w_B, with c_1 left of c_2. Finally, suppose that during the algorithm Put In Lexicographic Order, a subword of w_B with c_1 as its leftmost letter moves into w_A and lands to the immediate right of x_1, and that a subword of w_B with c_2 as its leftmost letter moves into w_A and lands to the immediate right of x_2. Then before either block is moved, there is an occurrence of a letter v between x_1 and x_2, and there are letters $d_k, d_{k-1}, \ldots, d_1 (k \geq 1)$ occurring in the indicated order (but possibly with other letters intermingled) between c_1 and c_2 such that the following hold:*

- $[v, d_k] \notin R$
- $[d_i, d_{i+1}] \notin R, i = 1, \ldots, k-1$
- $[d_1, c] \notin R$.

Note our slight abuse of notation in that the d_i are (possibly) distinct letters, whereas c_1 and c_2, as well as x_1 and x_2, are distinct occurrences of the letters c and x, respectively.

Proof. We first note several conditions which must hold before either block is moved.

1. In order for c_1 to move just right of x_1, c must commute with x and all letters between x_1 and x_2.

2. The letter, say z, just right of x_1 must be heavier than c lexicographically; otherwise c_1 would not move left of it. Similarly, the letter, say z', occurring just right of x_2 must also be heavier than c.

3. Since w_B is in lexicographic order, c_2 does not move left of x_2, and c commutes with every letter occurring between x_1 and x_2, there must be some letter $d \neq c$ occurring between c_1 and c_2 with which c does not commute and which lands between z and x_2. (If every letter occurring between c_1 and c_2 with which c does not commute lands to the left of z, then since c is lighter than z, c_2 would also land to the left of z.) Let d_1 be such a letter, and choose it to be the one which lands farthest to the right. Now $d_1 \neq x$, because c commutes with x but c does not commute with d_1. Also, since d_1 moves left of x_2, $[d_1, x] \in R$.

4. Let u be the leftmost letter in w_1 to the left of which d_1 moves; then u occurs between z and z'. Since d_1 moves left of u, d_1 and u must commute. And since c commutes with every letter between x_1 and x_2 and every letter which lands to the right of d_1, it must be the case that u and every letter between u and z' are lighter than c; otherwise c_2 would move left of x_2.

5. Since d_1 is lighter than u and u is lighter than c, d_1 is lighter than c lexicographically. Furthermore, c is lighter than z, so d_1 is also lighter than z.

Let D be the set of all chains $d_1, d_2, \ldots, d_k$, $k \geq 1$ of letters which occur in w_B in the order $d_k, d_{k-1}, \ldots, d_1$ such that $[c, d_1] \notin R$ and $[d_i, d_{i+1}] \notin R$, $i = 1, \ldots, k-1$. Note that d_1 is fixed, but $d_2, \ldots, d_k$ may represent different letters in different chains. (Since w_B has finite length,

this must be a finite set.) We claim that for some chain in D there is a letter v between x_1 and u which blocks d_k.

Consider any chain $d_1, d'_2, \ldots, d'_k$ in D. Since d_1 lands to the left of u and since no d'_i can move left of d'_{i+1}, every d'_i in the chain must move left of u. If every d_i in every chain is lighter than z, and there is no letter between x_1 and u blocking any d_k, then every d_i in every chain would move left of z, including d_1, which we have already seen cannot happen.

Now by way of contradiction suppose that there is no letter v occurring between x_1 and u blocking d_k for any chain in D. Then there must be some chain in D containing a letter d'_j, $j > 1$, such that d'_j is heavier than z: $d_1 < u < c < z < d'_j$. Since every letter in every chain in D moves left of u, there must be some letter t, which occurs between x_1 and u which is heavier than every letter in every chain in D. (It cannot be the case that $t = u$ because $u < d'_j < t$). Now by our assumption, there are no letters between x_1 and u which block d_k in any chain in D, and so by construction of D there are no letters between x_1 and u which block any d_i in any chain in D. Therefore every d_i in every chain in D moves left of t. Therefore d_1 moves left of t, contradicting our choice of u as the leftmost letter in w_1 to the left of which d_1 moves.

Therefore there is some letter v occurring between x_1 and u which blocks d_k for some chain in D. □

In the situation of the lemma above, let v be the rightmost letter occurring between x_1 and u which blocks d_k for some chain in D. Choose a shortest chain in D which is blocked by v and call this a *blocking chain* for the subword of w_A starting with x_1 and ending with x_2 and the subword of w_B starting with c_1 and ending with c_2. Let us refer to any pair of subwords where the first is a subword of w_A beginning and ending with x and the second is a subword of w_B beginning and ending with c, and where the first and second occurrences of c in the second subword land to the immediate right of the first and second occurrences, respectively,

of x in the second subword as an *xc-pair* of subwords. In general, if we do not know the beginning and ending letters of such a pair of subwords, we refer to the pair as a *bonding pair*, the letter with which the first subword begins and ends as the *left bond letter*, and the letter with which the second subword begins and ends as the *right bond letter*.

Lemma 2.49. *A chain is a blocking chain for only one xc-pair of subwords. Furthermore, if a chain beginning with d_k is a blocking chain for one xc-pair, then no chain beginning with d_k is a blocking chain for any other xc-pair.*

Proof. By Lemma 2.48, every bonding pair of subwords has a blocking chain. Suppose a chain beginning with d_k is a blocking chain for an xc-pair of subwords. That blocking chain cannot move left of the rightmost letter, say v, with which d_k does not commute, so v must lie between x_1 and x_2, the first and last letters of the first subword of the xc-pair. Any other xc-pair must lie left of that xc-pair, but that one letter v prevents any chain beginning with d_k from moving further left, so no other chain beginning with d_k will be able to move far enough left to act as the blocking chain for any other xc-pair. □

Lemma 2.50. *The number of subwords that move from the original w_B into the original w_A in the algorithm Put in Lexicographic Order 2 is no more than $m^3/2$.*

Proof. By Lemma 2.49, there is only one possible bonding pair of subwords having a blocking chain beginning with d_k. Suppose such a blocking chain exists and that it is a blocking chain for an xc-pair. Then $d_k \neq x$, since d_k moves left of x_2. Therefore there are no more than $m-1$ possible generators which can play the role of d_k for a given xc-pair, and so there are no more than $m-1$ xc-pairs in $w_A \cdot w_B$. Thus there are no more than m subwords beginning with c which land immediately right of an occurrence of x.

Now there are at most $m/2$ generators which can play the role of left bond letter in a bonding pair: Given two generators x' and c', either $x' < c'$ or $c' < x'$. If $x' < c'$ and $[x', c'] \in R$, then there can be $x'c'$-pairs, but no $c'x'$-pairs. If $[x', c'] \notin R$, then there are no $x'c'$- or $c'x'$-pairs;

there is at most one block in w_B beginning with either x' or c' which lands to the immediate right of a block in w_A beginning with either c' or x'. Once we have chosen a left bond letter, there are no more than $m-1$ generators which can serve as its right bond letter, since the left and right bond letters must be distinct. Thus there are $m(m-1)/2$ possible distinct bonding pairs of subwords in $w_A \cdot w_B$, where by distinct we mean that the left bond letters or the right bond letters are distinct.

Therefore, since for each distinct bonding pair there are no more than m subwords beginning with the right bond letter which land immediately right of an occurrence of the left bond letter, there are at most $m\left(\frac{m(m-1)}{2}\right) = \frac{m^3-m^2}{2}$ subwords which belong to bonding pairs which move from w_B into w_A in Put in Lexicographic Order 2.

Consider those subwords which move from w_B to w_A which do not belong to any bonding pair. Say there is only one subword beginning with c' which lands to the immediate right of an occurrence of x', so that there is no $x'c'$-pair. There are two cases: $[x', c'] \in R$ or $[x', c'] \notin R$. If x' and c' do not commute, then the leftmost occurrence of either in w_B will be blocked by the rightmost occurrence of either in w_A. By assumption the leftmost occurrence of c' in w_B is blocked by the rightmost occurrence of x' in w_A, so there is no subword of w_B beginning with x' which lands to the immediate right of an occurrence of c' in w_A. If, on the other hand, x' and c' do commute, then by our assumption, it must be the case that $x' < c'$. thus no occurrence of x' in w_B will land to the immediate right of an occurrence of c' in w_A. Thus, in either case, if there is only one subword beginning with c' in w_B which lands to the immediate right of an occurrence of x' in w_A, it is not possible for a subword in w_B beginning with x' to land to the immediate right of an occurrence of c' in w_A. Thus there are at most $m^2/2$ subwords not belonging to bonding pairs which move from w_B into w_A.

Together, then, there are at most $\frac{m^3-m^2}{2} + \frac{m^2}{2} = \frac{m^3}{2}$ subwords which move from w_B into w_A in Put in Lexicographic Order. □

For the following lemmas we will need several definitions. Let a subword that moves as a *cg*-block within the original w_B be called a *block*. Suppose that a subword C_0 of w_2 contains a letter with which the initial letter of a block C_1 fails to commute. Suppose further that C_0 is initially right of, but moves left of, some letter z_1 in w_B, and that after C_0 moves, without requiring any other blocks to move, C_1 moves to the immediate left of z_1. Then we say that C_0 *releases* C_1 to move within w_B. In this situation, we call the letter z_1 the *magnet letter* for the block C_1. We will generally use C_i to denote blocks, c_i to denote the initial letter of C_i, and z_i to denote the magnet letter of C_i.

For a given subword C_0 of w_B which moves into w_A, suppose that moving C_0 into w_B releases the block C_1 to move within w_B, and that for $i > 1$ moving C_{i-1} within w_B releases C_i to move within w_B. Then we call the sequence $C_1, C_2, \ldots, C_k$ a *branch* of blocks with *root* C_0 and *nodes* $C_1, C_2, \ldots, C_k$. Since w_B is of finite length, $k < \infty$. When we are discussing the blocks of a particular branch moving, there may be other letters in w_B to the left of, between, and to the right of the blocks. These letters do not move when the blocks of the given branch move; we call these *non-moving letters*, although they may belong to blocks for a different branch. For a given branch $C_1, C_2, \ldots, C_j, C_{j+1}, \ldots, C_k$ with root C_0, if there is another branch $C_1, C_2, \ldots, C_j, \overline{C_{j+1}}, \ldots, \overline{C_l}$ with root C_0, we say that C_j is a *branching node* of these two branches, that $C_1, C_2, \ldots, C_j$ is a *parent subbranch* of these two branches, and that these two branches *extend* from C_j.

Notice that by definition, and because w_B is originally in lexicographic order, no block in a given branch can move into or to the left of any block in that branch which lies to its left in w_B. Also, for a given branch, there must be at least one block between z_i and C_i for all i; otherwise w_B would not be in lexicographic order. And since C_i moves left of z_i, any blocks between z_i and C_i, as well as C_i itself, must move left of z_i and must therefore commute with z_i and any other non-moving letters between z_i and C_i. Furthermore, since w_B is in lexicographic order, any non-moving letters between some block C_{i-1} and C_i (the leftmost

block right of C_{i-1}) must be lighter than C_i in the lexicographic ordering.

Suppose blocks $C_1, C_2, \ldots, C_k$ lying in a given branch in the order indicated all have the same magnet letter; they all land between $z_1 = z_2 = \cdots = z_k$ (all representing the same occurrence of the same letter) and the letter immediately left of z_1 in w_B. Suppose further that the rightmost block left of C_1 and the leftmost block right of C_k each have a different magnet letter than $C_1, C_2, \ldots, C_k$. Then we call the sequence $\Lambda = C_1, C_2, \ldots, C_k$ a *cluster* of blocks and denote their magnet letter with ζ_i. Notice that because of our comment at the beginning of this paragraph, and by the definition of a cluster, $C_1, C_2, \ldots, C_k$ are consecutive blocks in the branch; there are no other blocks in the branch lying between C_1 and C_k. We will generally use Λ_i to denote clusters and ζ_i to denote their associated magnet letters. Because all of the blocks in Λ_i land to the left of ζ_i, ζ_i commutes with and is heavier than all of the letters in all of the blocks in Λ_i. Note that is possible for a cluster to consist of a single block. We will denote the initial block of a cluster Λ_i by $C_{i,1}$, the second block in Λ_i by $C_{i,2}$, and so on. The initial letter of $C_{i,1}$, and therefore of Λ_i, is denoted $c_{i,1}$.

Let $\Lambda_1, \Lambda_2, \ldots, \Lambda_r$ be consecutive clusters in a given branch such that $c_{1,1} < \zeta_1 < c_{2,1} < \zeta_2 < \cdots < c_{r,1} < \zeta_r$. Then we call the sequence $\Lambda_1, \Lambda_2, \ldots, \Lambda_r$ a *gang*. Suppose there is another gang $\Lambda'_1, \Lambda'_2, \ldots, \Lambda'_r$ to the right of $\Lambda_1, \Lambda_2, \ldots, \Lambda_r$ in the same branch such that $c'_{1,1} = c_{1,1}, c'_{2,1} = c_{2,1}, \ldots, c'_{r,1} = c_{r,1}$. Then we say that $\Lambda_1, \ldots, \Lambda_r$ and $\Lambda'_1, \ldots, \Lambda'_r$ are *similar* gangs. Notice that for two gangs in a given branch to be similar, the only requirement is that the initial letter of the initial block of each cluster in one gang is the same as the initial letter of the initial block of each corresponding cluster in the second gang.

We have already shown that the number of blocks which move from w_B into w_A is bound by $m^3/2$. In order to show a constant bound on the number of subwords of w_B which move within w_B, we will begin by showing a constant bound on the number of blocks in a given branch and then proceed to show a limit on the number of branches for a given root. The following two lemmas give us building blocks to be used in the third lemma below.

Lemma 2.51. *For any sequence $\Lambda_1, \Lambda_2, \ldots, \Lambda_r$ of consecutive clusters such that ζ_r lies to the left of Λ_1, $r \leq m^2/2$.*

Proof. By definition, Λ_i and Λ_{i+1} land separately for all i in $\{1, 2, \ldots, r\}$. So for any given pair Λ_i, Λ_{i+1}, either $\zeta_i < c_{i+1,i}$ or there is a letter k_{i+1} between ζ_i and ζ_{i+1}, or which is ζ_i itself, such that k_{i+1} commutes with $c_{1,1}, c_{2,1}, \ldots, c_{i,1}$ and k_{i+1} blocks $c_{i+1,1}$. Let us refer to any such k_i as a *blocker*. Thus for any $i, j \in \{1, 2, \ldots, r-1\}$ such that $i \neq j$ and blockers k_{i+1}, k_{j+1} exist, k_{i+1} and k_{j+1} are distinct, and the three initial letters $c_{i,1}$, $c_{i+1,1}$, and $c_{j+1,1}$ are distinct. Therefore there are no more than m initial letters $c_{i,1}$ in the sequence of clusters which fail to commute with a blocker, and there are no more than $m-1$ blockers for the clusters in the sequence.

Suppose k_{l_1} and k_{l_2} are two blockers and there are no blockers between k_{l_1} and k_{l_2}. Then k_{l_1} blocks $c_{l_1,1}$ and k_{l_2} blocks $c_{l_2,1}$. Since there are no blockers in between these two, we must have $\zeta_i < c_{i+1,1}$ for all $i \in \{l_1, l_1+1, \ldots, l_2-1\}$. Therefore, since every magnet letter is heavier than its associated cluster, the compound inequality $c_{l_1,1} < \zeta_{l_1} < c_{l_1+1,1} < \zeta_{l_1+1} < \cdots < c_{l_2-1,1} < \zeta_{l_2-1}$ holds. Thus no more than $m/2$ clusters land between k_{l_1} and k_{l_2}. Similarly, no more than $m/2$ clusters in the sequence of clusters land left of the leftmost blocker, and no more than $m/2$ clusters in the sequence of clusters land to the right of the rightmost blocker. It follows that there are no more than $m/2 + (m-2)m/2 + m/2 = m^2/2$ clusters in the sequence. □

Lemma 2.52. *For any cluster of blocks $C_1, C_2, \ldots, C_s$, $s \leq (m^2+m)/2$.*

Proof. By definition, $C_1, C_2, \ldots, C_s$ land together. Since they are separate blocks, therefore, each pair of consecutive blocks must be separated by at least one non-moving letter before moving. Recall that all of the letters in each C_i must commute with the magnet letter ζ and every non-moving letter between ζ and itself. Thus for every $i \in \{1, 2, \ldots, r-1\}$, either there is a letter p_{i+1} between C_i and C_{i+1} such that $c_i < p_{i+1} < c_{i+1}$ or there is a letter q_{i+1}

between C_i and C_{i+1} such that $c_i > q_{i+1} < c_{i+1}$ and q_{i+1} fails to commute with some letter in C_i. Let us refer to such p_{i+1} as *weight separators* and such q_{i+1} as *blocking separators*.

Let q_{i+1} and q_{j+1} be two blocking separators for blocks in the cluster $C_1, C_2, \ldots, C_s$, with $i < j$. Then q_{i+1} blocks some letter in C_i and q_{j+1} blocks some letter in C_j, but since C_i and C_j land together, q_{i+1} must commute with every letter in C_j. Hence q_{i+1} and q_{j+1} are distinct. Since this holds for any two blocking separators for blocks in the given cluster, there can be no more than m blocking separators between C_1 and C_s.

Let q_{l_1} and q_{l_2} be two blocking separators for the given cluster with no blocking separators between them. Then for every $i \in \{l_1, l_1 + 1, \ldots, l_2 - 1\}$ we must have a weight separator p_{i+1} between C_i and C_{i+1}, so $c_i < p_{i+1} < c_{i+1}$. This means that $q_{l_1} < c_{l_1} < p_{l_1+1} < c_{l_1-1} < \cdots < p_{l_2-1} < c_{l_2-1}$. Thus there are no more than $m/2$ blocks in the given cluster between q_{l_1} and q_{l_2}. Similarly, there are no more than $m/2$ blocks in the given cluster which lie left of the leftmost blocking separator and no more than $m/2$ blocks in the cluster which lie right of the rightmost blocking separator. All together, then, there are no more $m/2 + (m-1)m/2 + m/2 = (m^2 - m)/2$ blocks in the cluster $C_1, C_2, \ldots, C_s$. □

Lemma 2.53. *There are no more than $m^4/4 - m^3/2$ clusters in any given branch of w_B.*

Proof. Fix a branch with root C_0 and a gang $\Lambda_1, \ldots, \Lambda_r$ in that branch. Suppose there are exactly θ other gangs in the branch which are similar to $\Lambda_1, \ldots, \Lambda_r$, and that $\Lambda_1, \ldots, \Lambda_r$ is the leftmost of all $\theta + 1$ of these similar gangs. Denote the second leftmost of these gangs by $\Lambda'_1, \ldots, \Lambda'_r$, the third leftmost by $\Lambda''_1, \ldots, \Lambda''_r$, and so on, with the rightmost of these gangs being $\Lambda_1^{(\theta)}, \ldots, \Lambda^{(\theta)}$. We first provide a bound on the number of clusters between and including Λ_1 and $\Lambda_r^{(\theta)}$.

Let $C_{s,t}$ be the leftmost block in the branch right of the magnet letter $\zeta_r^{(\theta)}$. Then $\zeta_r^{(\theta)} < c_{s,t}$, so we have $c_{1,1} < \zeta_1 < c_{2,1} < \zeta_2 < \cdots < c_{r,1} = c_{r,1}^{(\theta)} < \zeta_r^{(\theta)} < c_{s,t} < \zeta_s$. Now by the definition of magnet letter, $\zeta_r^{(\theta)}$ must lie left of $\Lambda_r^{(\theta)}$. Furthermore, it is not possible for

$\zeta_r^{(\theta)}$ to lie between two blocks in one of clusters in one of the gangs: If $\zeta_r^{(\theta)}$ lies between $C_{i,j}^{(\alpha)}$ and $C_{i,j+1}^{(\alpha)}$ for some $i \in \{1, 2, \ldots, r\}$, where $\alpha \in \{0, 1, \ldots, \theta\}$, then we would have $\zeta_r^{(\theta)} < c_{i,j+1} < \zeta_i < c_{i+1,1} < \zeta_{i+1} < \cdots < c_r = c_r^{(\theta)} < \zeta_r^{(\theta)}$, which is not possible. Similarly, if $\zeta_r^{(\theta)}$ lies between two clusters $\Lambda_{i-1}^{(\alpha)}$ and $\Lambda_i^{(\alpha)}$ in one of the gangs, then $\zeta_r^{(\theta)} < c_{i+1,1} < \zeta_{i+1} < c_{i+2,1} < \zeta_{i+2} < \cdots < c_r = c_r^{(\theta)} < \zeta_r^{(\theta)}$, which is also a contradiction. Thus $\zeta_r^{(\theta)}$ lies either to the left of Λ_1 or between $\Lambda_r^{(\alpha)}$ and $\Lambda_0^{(\alpha+1)}$ for some $\alpha \in \{0, 1, \ldots, \theta - 1\}$, where $\Lambda_0^{(\alpha+1)}$ is the rightmost cluster left of $\Lambda_1^{(\alpha+1)}$.

Therefore, by definition of $C_{s,t}$, $C_{s,t}$ lies either left of Λ_1 or between $\Lambda_r^{(\alpha)}$ and $\Lambda_1^{(\alpha+1)}$. If $C_{s,t}$ lies left of Λ_1, then $\zeta_r^{(\theta)}$ is also left of Λ_1, and so by Lemma 2.51, the number of clusters between and including Λ_1 and $\Lambda_r^{(\theta)}$ in the given branch is no more than $m^2/2$. Otherwise, $C_{s,t}$ is right of Λ_r, and since $\zeta_r^{(\theta)}$ is left of $C_{s,t}$, the number of clusters in the branch between and including $C_{s,t}$ and $\Lambda_r^{(\theta)}$ is bounded by $m^2/2$.

Let C_{j_1,k_1} be the leftmost block which lies to the right of ζ_s, the magnet letter for $C_{s,t}$. Then $\zeta_s < c_{j_1,k_1} < \zeta_{j_1}$, so combining that with the inequality involving ζ_s above gives us $c_{1,1} < \zeta_1 < c_{2,1} < \zeta_2 < \cdots < c_{r,1} = c_{r,1}^{(\theta)} < \zeta_r^{(\theta)} < c_{s,t} < \zeta_s < c_{j_1,k_1} < \zeta_{j_1}$. Similar to the case for $\zeta_r^{(\theta)}$ above, ζ_s is either left of Λ_1 or between $\Lambda_r^{(\alpha)}$ and $\Lambda_0^{(\alpha+1)}$ for some $\alpha \in \{0, 1, \ldots, \theta - 1\}$. If ζ_s lies left of Λ_1, then by Lemma 2.51, the number of clusters in the given branch between and including Λ_1 and $C_{s,t}$ is no more than $m^2/2$. Combining this with the bound on the number of clusters between and including $C_{s,t}$ and $\Lambda_r^{(\theta)}$, we see that there are no more than $2m^2/2 = m^2$ clusters between and including Λ_1 and $\Lambda_r^{(\theta)}$ in the branch. If ζ_s is to the right of Λ_r, then since ζ_s is left of C_{j_1,k_1}, there are, by Lemma 2.51, no more than $m^2/2$ clusters in the given branch between C_{j_1,k_1} and $C_{s,t}$, including the cluster containing C_{j_1,k_1}. Therefore, since the number of clusters between and including $C_{s,t}$ and $\Lambda_r^{(\theta)}$ is no more than $m^2/2$, there are at most $2m^2/2 = m^2$ clusters between and including C_{j_1,k_1} and $\Lambda_r^{(\theta)}$.

We proceed to show by induction that for all $i > 0$, if we let $C_{j_{i+1},k_{i+1}}$ be the leftmost block right of ζ_{j_i} in the given branch, then the following hold:

1. $c_{1,1} < \zeta_1 < c_{2,1} < \zeta_2 < \cdots < c_{r,1} = c_{r,1}^{(\theta)} < \zeta_r^{(\theta)} < c_{s,t} < \zeta_s < c_{j_1,k_1} < \zeta_{j_1} < \cdots < c_{j_{i+1},k_{i+1}} < \zeta_{j_{i+1}}$;

2. Either $C_{j_{i+1},k_{i+1}}$ lies between $\Lambda_r^{(\alpha)}$ and $\Lambda_1^{(\alpha+1)}$ for some $\alpha \in \{0, 1, \ldots, \theta - 1\}$ or $C_{j_{i+1},k_{i+1}}$ lies left of Λ_1;

3. If $C_{j_{i+1},k_{i+1}}$ lies left of Λ_1, then the number of clusters in the given branch between and including Λ_1 and $\Lambda_r^{(\theta)}$ is no more than $(i+1)m^2/2$; and

4. If $C_{j_{i+1},k_{i+1}}$ is not left of Λ_1, then there are at most $(i+1)m^2/2$ clusters between and including $C_{j_{i+1},k_{i+1}}$ and $\Lambda_r^{(\theta)}$ in the given branch.

Suppose these hypotheses hold for all $i \in \{1, 2, \ldots, u-1\}$ for some $u > 0$. We will show that they hold for u as well. We let $C_{j_{u+1},k_{u+1}}$ be the leftmost block right of ζ_{j_u}, and so $\zeta_{j_u} < c_{j_{u+1},k_{u+1}} < \zeta_{j_{u+1}}$. Combining this with the inequality above for $i = u - 1$ gives $c_{1,1} < \zeta_1 < c_{2,1} < \zeta_2 < \cdots < c_{r,1} = c_{r,1}^{(\theta)} < \zeta_r^{(\theta)} < c_{s,t} < \zeta_s < c_{j_1,k_1} < \zeta_{j_1} < \cdots < c_{j_u,k_u} < \zeta_{j_u} < c_{j_{u+1},k_{u+1}} < \zeta_{j_{u+1}}$.

Now it is not possible for ζ_{j_u} to lie between two blocks in one of clusters in one of the gangs: If ζ_{j_u} lies between $C_{v,w}^{(\alpha)}$ and $C_{v,w+1}^{(\alpha)}$ for some $v \in \{1, 2, \ldots, r\}$, where $\alpha \in \{0, 1, \ldots, \theta\}$, then $\zeta_{j_u} < c_{v,w+1}^{(\alpha)} = c_{v,w+1}$. Thus by the inequality just shown, we would have $\zeta_r^{(\theta)} < c_{j_u,k_u} < \zeta_{j_u} < c_{v,w+1} < \zeta_v < c_{v+1,1} < \zeta_{v+1} < \cdots < c_r = c_r^{(\theta)} < \zeta_r^{(\theta)}$, which is not possible. Similarly, if ζ_{j_u} lies between two clusters $\Lambda_{v-1}^{(\alpha)}$ and $\Lambda_v^{(\alpha)}$ in one of the gangs, then $\zeta_r^{(\theta)} < c_{j_u,k_u} < \zeta_{j_u} < c_{v,1} < \zeta_v < c_{v+1,1} < \zeta_{v+1} < \cdots < c_r = c_r^{(\theta)} < \zeta_r^{(\theta)}$, which is also a contradiction. Thus ζ_{j_u} lies either to the left of Λ_1 or between $\Lambda_r^{(\alpha)}$ and $\Lambda_0^{(\alpha+1)}$ for some $\alpha \in \{0, 1, \ldots, \theta - 1\}$. Hence, by our definition of $C_{j_{u+1},k_{u+1}}$, $C_{j_{u+1},k_{u+1}}$ is either left of Λ_1 or between $\Lambda_r^{(\alpha)}$ and $\Lambda_1^{(\alpha+1)}$ for some α.

If ζ_{j_u} lies left of Λ_1, then by Lemma 2.51, the number of clusters in the given branch between and including Λ_1 and C_{j_u,k_u} is no more than $m^2/2$. Combining this with the bound $um^2/2$ on the number of clusters between and including C_{j_u,k_u} and $\Lambda_r^{(\theta)}$ given in the induction

hypothesis, we see that there are no more than $um^2/2+m^2/2 = (u+1)m^2/2$ clusters between and including Λ_1 and $\Lambda_r^{(\theta)}$ in the branch. If ζ_{j_u} is not left of Λ_1, then since ζ_{j_u} is left of $C_{j_{u+1},k_{u+1}}$, there are, by Lemma 2.51, no more than $m^2/2$ clusters in the given branch between $C_{j_{u+1},k_{u+1}}$ and C_{j_u,k_u}, including the cluster containing $C_{j_{u+1},k_{u+1}}$. Therefore, since the number of clusters between and including C_{j_u,k_u} and $\Lambda_r^{(\theta)}$ is no more than $um^2/2$, there are at most $(u+1)m^2/2$ clusters between and including $C_{j_{u+1},k_{u+1}}$ and $\Lambda_r^{(\theta)}$.

Now that we know that the conditions above hold for all $i > 0$, we are able to complete the proof. Notice that the inequality $c_{1,1} < \zeta_1 < c_{2,1} < \zeta_2 < \cdots < c_{r,1} = c_{r,1}^{(\theta)} < \zeta_r^{(\theta)} < c_{s,t} < \zeta_s < c_{j_1,k_1} < \zeta_{j_1} < \cdots < c_{j_x,k_x} < \zeta_{j_x}$ is an inequality containing $2r+2+2x$ distinct letters. We have only m distinct letters to use, so $2r+2+2x \leq m$, which means that $x \leq m/2-r-1$. Therefore ζ_{j_i} must lie left of Λ_1 for some $i \leq m/2-r-1$, and it follows that there are at most $\frac{[(m/2-r-1)+1]m^2}{2} = \frac{m^3}{4} - \frac{m^2 r}{2}$ clusters between and including Λ_1 and $\Lambda_r^{(\theta)}$ in the given branch. Now $r \geq 1$, so there are no more than $m^3/4 - m^2/2$ clusters between and including Λ_1 and $\Lambda_r^{(\theta)}$ in the given branch.

Consider now the number of possible collections of gangs. We know that there are no more gangs similar to $\Lambda_1, \ldots, \Lambda_r$ than the $\theta+1$ gangs we discussed above. In fact, by allowing a gang to consist of a single cluster Λ_1, we have taken into account all gangs beginning with $c_{1,1}$. There are m distinct letters, so there are at most m collections of similar gangs which begin with distinct letters. Since every cluster is part of a gang, therefore, there are no more than $m(m^3/4 - m^2/2) = m^4/4 - m^3/2$ clusters in the given branch. □

Lemma 2.54. *The total number of subwords that move when the algorithm Put in Lexicographic Order 2 is run once is no more than* $\frac{m^3}{2}\left[1 + \sum_{i=0}^{(m^6-2m^5-m^4)/8} m^i\right]$.

Proof. Combining Lemmas 2.52 and 2.53, we see that there are no more than $\frac{m^2+m}{2} \cdot \frac{m^4-2m^3}{4} = \frac{m^6-2m^5-m^4}{8}$ blocks in any branch in w_B. Next we consider the number of possible branches.

A given subword C_0 can release at most m blocks to move within w_B: Suppose C_0 releases a block C_1 to move within w_B. Let C_1' be any block right of C_1 with the same initial letter as C_1, $c_1 = c_1'$. Then there is some letter between c_1 and c_1' with which c_1' does not commute; otherwise since w_B is in lexicographic order, c_1' would be immediately to the right of c_1 and thus be in the same block as c_1. Therefore C_0 cannot release C_1'; the letter between c_1 and c_1' with which c_1' does not commute cannot lie in C_0, which lies left of C_1. Therefore any subword can release at most one block with a given initial letter.

This means that the root and every node in a branch can branch at most m times. So the root has no more than m parent subbranches extending from it; each initial node of these parent subbranches has at most m branches extending from it; and so on. Since no branch has more than $(m^6 - 2m^5 - m^4)/8$ nodes, this means that there are at most $\sum_{i=0}^{(m^6-2m^5-m^4)/8} m^i$ branches total from any given root.

By Lemma 2.50, there are no more than $m^3/2$ subwords which move from w_B into w_A; that is, there are no more than $m^3/2$ roots in w_B. Therefore there are at most $\frac{m^3}{2} \sum_{i=0}^{(m^6-2m^5-m^4)/8} m^i$ blocks in w_B.

Adding to this the number of possible roots, there are at most

$$\frac{m^3}{2} + \frac{m^3}{2} \sum_{i=0}^{(m^6-2m^5-m^4)/8} m^i = \frac{m^3}{2} \left[1 + \sum_{i=0}^{(m^6-2m^5-m^4)/8} m^i \right]$$

subwords in w_B which move, either into w_A or within w_B. □

Lemma 2.55. *The algorithm Put In Lexicographic Order 2 described by the flowchart in Figure 2.25 runs in polynomial time in $n + p$, where n and p are the lengths of the SLPs which are input.*

Proof. We will first see that the number of steps required for each iteration of the large outer loop is polynomial in $n + p$, and then we will show that the number of iterations of the outer

loop is bounded by a polynomial in $n + p$ as well.

The time it takes to run the algorithm Find First Occurrence Order is polynomial in p by Lemma 2.45, say $q_1(p)$ steps, and assigning values to variables before entering the main loop happens in constant time, say c_1 steps. So before entering the main loop, the time required is $q_1(p) + c_1$.

Checking to see whether or not $i < m$ and assigning values to the variables c and $CanMove$ are done in constant time, say c_2 steps. Count the Occurrences runs in polynomial time in p, say $q_2(p)$ steps, by Lemma 2.19. So for each iteration of the main loop, $q_2(p) + c_2$ steps are required before entering the *numcs* loop.

Consider the part of the routine shown in Figure 2.26, which happens at the beginning of the *numcs* loop. The routines Find the Occurrence and Left Sub SLP both run in polynomial time in p by Lemmas 2.29 and 2.23. Now by Lemma 2.24, the length of the SLP $\mathbb{L}$ofc produced by Left Sub SLP is no more than $2p$. This is the SLP output to Find Included Generators, so by Lemma 2.7, Find Included Generators runs in polynomial time in $2p$. A polynomial in $2p$ is a polynomial in p, and the sum of three polynomials in p is a polynomial in p, so these three routines together run in polynomial time in p, say $q_3(p)$ steps. The little loop which checks to see if a_c commutes with all of the generators in **LofcGens** has m or fewer iterations, and the steps in each iteration are bounded by a constant, so the total number of steps required by the loop is bounded by a constant. The other operations which occur in the part of the *numcs* loop illustrated in 2.26 are also bounded by a constant. Thus the part of the routine shown in 2.26 is done in $q_3(p) + c_3$ steps.

Now checking to see if $CanMove = true$ happens in constant time; let us include this time in our calculation of the number of steps required by the part of the *numcs* loop illustrated in Figure 2.27. By Lemmas 2.35, 2.5, and 2.26, the algorithms Find Rightmost Noncommuting, Get Length, and Right Sub SLP each run in polynomial time in n, say $q_4(n)$ steps altogether.

The other operations in the part of the *numcs* loop shown in Figure 2.27 happen in constant time, so the part of the routine shown in Figure 2.27 runs in $q_4(n) + c_4$ steps.

By Lemma 2.27, the length of $\mathbb{R}$ofb is no more than $2n$, so by Lemma 2.40, when we apply the algorithm Find Leftmost Heavier to $\mathbb{R}$ofb, it runs in polynomial time in $2n$, and thus in polynomial time in n, say $q_5(n)$. Determining whether or not $zpos = -1$ happens in constant time, say c_5 steps, so this little part of the routine that happens between the parts shown in Figures 2.27 and 2.28 requires $q_5(n) + c_5$ steps.

Consider the part of the *numcs* loop shown in Figure 2.28. Right Sub SLP runs in polynomial time in n and produces an SLP of length $2n$ or less. Thus Find Included Generators runs in polynomial time in $2n$, which is polynomial in n. These two algorithms together therefore run in polynomial time in n, say $q_6(n)$ steps. The other operations that happen in the part illustrated in Figure 2.28 happen in constant time, say c_6 steps. Hence $q_6(n) + c_6$ is the number of steps for the part in Figure 2.28.

The steps required to create the list **genlist** run in constant time since the little loop runs m times, and none of the steps in the loop depend on n or p. Let us include this time in the number of steps taken in the part of the *numcs* loop pictured in Figure 2.29. The algorithms Get Length and Right Sub SLP each run in polynomial time in p, and by Lemmas 2.27, 2.37, and 2.23, Find Leftmost Noncommuting - List and Left Sub SLP run in polynomial time in $2p$. Furthermore, Lemmas 2.24, 2.41, and 2.23 imply that Find Leftmost Heavier and Left Sub SLP each run in polynomial time in $4p$. For future reference, note that the length of $\mathbb{C}\mathbb{G}$Block is no more than $8p$. Altogether then, these six routines run in polynomial time in p, say $q_7(n)$ steps. The operations that happen outside of these routines in the part shown in Figure 2.29 run in constant time, say c_7 steps. Therefore this part of the *numcs* loop runs in $q_7(p) + c_7$ steps.

We continue to the part illustrated in Figure 2.30. As we saw earlier, the length of $\mathbb{R}$ofb is

no more than $2n$; thus Find Included Generators runs in polynomial time in $2n$ and hence in n. Now $\mathbb{CG}$Block has $8p$ or fewer production rules, so Find Leftmost Commuting - List runs in polynomial time in $8p$ and thus in p. Adding a polynomial in n to one in p gives a polynomial which is bounded by a polynomial in $n+p$; say $q_8(n+p)$ is a bound for the number of steps in these two algorithms together. The other operations outside of the f loop run in constant time, say c_8 steps, so the number of steps in the part in Figure 2.30 outside of the f loop is $q_8(n+p)+c_8$.

We now enter the f loop. For each iteration of the f loop, we begin by applying Left Sub SLP to $\mathbb{CG}$Block; this runs in polynomial time in $8p$ and so in p, and it produces an SLP $\mathbb{C}$f1Block of length $16p$ or less. Therefore Find Included Generators runs in polynomial time in $16p$, so in p. The steps required to create the list **chkGList** run in constant time since the little loop runs m times, and none of the steps in the loop depend on n or p. When we output $\mathbb{CG}$Block to Right Sub SLP, the routine runs in polynomial time in $8p$, so in p, and returns an SLP $\mathbb{F}$gBlock of length no more than $16p$. Hence Find Leftmost Commuting - List runs in polynomial time in $16p$, so in p. Let us say that these routines together require $q_9(p)$ steps. The other operations inside the f loop which happen outside of the part illustrated in Figure 2.31 run in constant time, say c_9 steps. Thus, outside of the part shown in Figure 2.31, each iteration of the f loop runs in $q_9(p)+c_9$ steps.

Similarly, in the part shown in Figure 2.31, the first four routines called, Left Sub SLP, Find Rightmost Heavier, Right Sub SLP, and Find Included Generators, together run in polynomial time in n. The last two routines called, Right Sub SLP, and Find Leftmost Commuting - List run in polynomial time in p. The operations outside these routines in the part shown in Figure 2.31 run in constant time. Therefore the number of steps required for the part in this flowchart is bounded by a polynomial in $n+p$, say $q_{10}(n+p)$.

Recall that in the f loop we determine whether or not there is an a_f in the cg-block which can and should move left of a_b. We find our first a_f that may end up moving left of a_b by finding

the leftmost letter in the *cg*-block which commutes with a_b. If we determine that a_f will not end up moving left of a_b, we append f to **chkGList** and find the next a_f by finding the leftmost letter in the *cg*-block lying to the right of the previous a_f which commutes with all of the generators in **chkGList**. Now there are at most $m-1$ generators which can commute with a_b (recall that a_b does not commute with itself), which is in **chkGList**, and we append each f to **chkGList** before finding the next a_f, which must therefore commute with the previous a_f and so be a different generator than any previous a_f. Therefore there are at most $m-1$ letters in the *cg*-block which will get assigned to the variable a_f. Thus the f loop has no more than $m-1$ iterations, and so running the f loop requires $(m-1)[q_9(p)+c_9+q_{10}(n+p)+c_10]$ steps, which is bounded by a polynomial in $n+p$. Add to this bound the number of steps occurring in Figure 2.30 outside of the f loop – that is, $q_8(n+p)+c_8$ – to get that the number of steps required for the part of the *numcs* loop in Figure 2.30 is bounded by a polynomial in $n+p$, say $q_{11}(n+p)$.

The algorithm Combine SLPs runs in polynomial time in the sum of the lengths of the two SLPs which are input, by Lemma 2.21. So, for the same reasons as with other parts of the *numcs* loop, the algorithms called in the part of the routine represented in Figure 2.33 together run in polynomial time in $n+p$, say $q_{12}(n+p)$ steps. Similarly, the three algorithms and one other operation which happen in the part illustrated in Figure 2.34 together run in polynomial time in p, say $q_{13}(p)$ steps.

For reasons we will soon explain, the total number of times that the *numcs* loop can be run during the entire algorithm Put In Lexicographic Order 2 (not only in a single iteration of the main loop, or in a single iteration of the large outer loop, but in all of the iterations of these loops together) is bounded by a constant; let us call it C. This means that the product of the number of iterations of the main loop and the number of iterations of the outer loop is bounded by C, and so the number of iterations of the outer loop is also no more than C.

Therefore the total number of steps occurring in the *numcs* loop for the entire duration of

the algorithm is no more than $C[q_3(p) + c_3 + q_4(n) + c_4 + q_5(n) + c_5 + q_6(n) + c_6 + q_7(p) + c_7 + q_8(n+p) + c_8 + q_9(p) + c_9 + q_{10}(n+p) + q_{11}(n+p) + q_{12}(n+p) + q_{13}(p)]$, which is bounded by a polynomial in $n+p$, say $Q_1(n+p)$. The total number of steps occurring in the main loop outside of the *numcs* loop for the entire duration of the algorithm is no more than $C[q_2(p) + c_2] = Q_2(p)$, and the total number of steps occurring in the outer loop outside of the main loop for the entire duration of the algorithm is no more than $C[q_1(p) + c_1] = Q_3(p)$. The sum $Q_1(n+p) + Q_2(p) + Q_3(p)$ is the total number of steps in the entire algorithm, and it is bounded by a polynomial in $n+p$. It remains to for us to show that the total number of times the *numcs* loop is run during the entire duration of the algorithm is bounded by a constant; we address this next.

By Lemma 2.54, there are no more than $\frac{m^3}{2}\left[1 + \sum_{i=0}^{(m^6-2m^5-m^4)/8} m^i\right]$ *cg*-blocks which move during one run of Put in Lexicographic Order 2. Let us call this constant C'. Every time the *numcs* loop is run, either it is determined that no block beginning with a_c will move into w_X, in which case we exit back out to the main loop, or a *cg*-block will move. Thus for a single iteration of the main loop, there is only one iteration of the *numcs* loop for which a *cg*-block does not move. For every iteration of the large outer loop there are no more than m iterations of the main loop, so for every iteration of the large outer loop, there are no more than m iterations of the *numcs* loop for which a *cg*-block does not move. The large outer loop is repeated only as long as at least one *cg*-block moves during the previous iteration. For every *cg*-block that moves, therefore, there must be no more than m iterations of the *numcs* loop for which a *cg*-block does not move, if we do not include the last iteration of the large outer loop. Thus there are no more than $mC' + m$ iterations of the *numcs* loop altogether, where adding m at the end takes into account the last iteration of the large outer loop. Since $C = mC' + m$ is a constant, the number of times the *numcs* loop is run during the entire duration of the algorithm is bounded by a constant, and this completes our proof. We note that this proof also implies that the length of the SLP output by Put in Lexicographic Order 2 is also a polynomial in $n+p$. □

The algorithm we discuss next, Put in Lexicographic Order, takes an SLP $\mathbb{A}$ and returns an SLP which produces the word which similar to w_A but in lexicographic order. In order to solve the word problem for the automorphism group of a RAAG, it is not necessary to put a word into shortlex form; we only need to put a $\phi(a_i)$ into shortest form to see if it equals a_i. However, we use lexicographic ordering in order to put words into their shortest forms. There may be ways of putting a word into shortest form without using lexicographic ordering, but our way has the additional benefit of providing a normal form for words and a polynomial-time algorithm for finding that normal form.

Lemma 2.56. *The algorithm Put in Lexicographic Order described by the flowchart in Figure 2.35 inputs an SLP $\mathbb{A}$, an index u, and an indicator fr, set to either 'f' or 'r', and it returns an SLP in quasinormal form which produces the word which is similar to w_A and in forward (if $fr =$ 'f') or reverse (if $fr =$ 'r') lexicographic order, where a_u is set to be the heaviest generator and for all $i \in \{0, 1, \ldots, u-2, u+1, ldots, m\}$, $a_i < a_{i+1}$.*

Proof. After inputting $\mathbb{A}$, u, and fr, although it is not indicated in the flowchart, we find the length of w_A, and if $|w_A| < 2$, we return $\mathbb{A}$ and exit the routine. Otherwise, we create a list **Genord** of length m with each item set to the value $m-1$ and initialize the variable $\mathbb{S}2O$, short for 'SLP to order', to $\mathbb{A}$. Each item **Genord**$[j]$ in **Genord** will set to the position of a_j in the ordering; the purpose of the first loop is to accomplish this.

We use i to step through the generators one at a time, starting with $i = 0$. For each iteration of this first little loop, we check to see if $i = u$. If not, we set **Genord**$[i]$ to q and increase q by 1. If $i = u$, we do nothing before returning to the beginning of the loop to check the next generator. Thus **Genord**$[u]$ remains set to its initial value of $m-1$, but all of the other generators are weighted consecutively. This way **Genord** contains the lexicographic ordering in which a_u is the heaviest generator and for all $i \in \{0, 1, \ldots, u-2, u+1, ldots, m\}$, $a_i < a_{i+1}$.

Before beginning the main loop, we check to see if $fr =$ 'r', and if so, we output $\mathbb{A}$ to the

algorithm Reverse SLP, which by Lemma 2.16 returns an SLP producing the reverse of w_A. We reassign $\mathbb{S}2O$ to this SLP. If $fr \neq$ 'r' then there is no need to do anything to $\mathbb{S}2O$ at this point. We also reset q to 0, let n be the number of production rules in $\mathbb{S}2O$, and create a list **SLPList** of length n with each item set to 0 before entering the main loop.

For the main loop, we use q to step through the the production rules of $\mathbb{S}2O$ one at a time, starting with $q = 0$. For simplicity, let us denote the nonterminal characters of $\mathbb{S}2O$ by S_i. We begin each iteration of the main loop by checking to see if the current production rule is of the form $S_q \to S_r \cdot S_s$. If not, we return to the beginning of the main loop to continue with the next production rule. If the current production rule is of the form $S_q \to S_r \cdot S_s$, we consider **SLPList**. If **SLPList**$[r] = 0$, then we have not changed its value yet, so we know that the production of S_r is a letter. Hence we output the ProdRule for S_r to Letter to SLP and set **SLPList**$[r]$ to the SLP which is returned. By Lemma 2.8, this SLP produces the letter $w(S_r)$. Similarly, if **SLPList**$[s] = 0$, we output the ProdRule for S_s to Letter to SLP and set **SLPList**$[s]$ to the SLP which is returned, which produces the letter $w(S_s)$. If **SLPList**$[r] \neq 0$, then we do not change its value; similarly we leave **SLPList**$[s]$ unchanged if its value is not 0.

Whether or not we changed the value of **SLPList**$[r]$ or **SLPList**$[s]$, we set the variables $SLPr$ and $SLPs$ to **SLPList**$[r]$ and **SLPList**$[s]$, respectively. We then output $SLPr$, $SLPs$, and **Genord** to Put in Lexicographic Order 2, which by Lemma 2.46 returns an SLP producing a lexicographically ordered word similar to $w(S_r) \cdot w(S_s)$, according to the ordering given in **Genord**. We then return to the beginning of the main loop to deal with the next production rule. Hence, at the end of each step of the main loop, **SLPList**$[q]$ contains an SLP in quasinormal form which produces the lexicographically ordered word similar to $w(S_q)$. Therefore, after completing the main loop, the last entry in **SLPList**, that is, **SLPList**$[n-1]$, contains an SLP in quasinormal form which produces the lexicographically ordered word similar to w_S. So after completing the main loop, we set $\mathbb{O}$rdSLP to **SLPList**$[n-1]$ and $\mathbb{B}$ to $\mathbb{O}$rdSLP.

We must check again to see if $fr =$ 'r'. If so, we again run Reverse SLP, this time on $\mathbb{O}$rdSLP, and set $\mathbb{B}$ to the SLP which is returned. In this case, since we reversed the word, ordered it lexicographically, and reversed it again, w_B is similar to w_A but in reverse lexicographic order, that is, with the lightest letters as far right as possible instead of as far left as possible.

Finally, we run the algorithm Quasinormalize SLP on $\mathbb{B}$, so the new SLP is in quasinormal form by Lemma 2.10, and the word produced by this new SLP is similar to the word produced by $\mathbb{A}$ but in forward or reverse (if and only if $fr =$ 'r') lexicographic order, with a_u given the heaviest weight. □

Lemma 2.57. *The algorithm Put In Lexicographic Order described by the flowchart in Figure 2.35 runs in polynomial time in n, the length of the SLP which is input.*

Proof. The operations that happen before the first loop, as well as the time required to determine whether or not $i < n$, do not depend on n and so happen in constant time, say c_1 steps. For each iteration of the first loop, the time is independent of n, so the number of steps is bounded by a constant, say c_2. Running Reverse SLP the first time happens in polynomial time in n, say $q(n)$ by Lemma 2.17. The other steps that occur before entering the main loop happen in constant time, say c_3 steps. The first loop runs m times, so the number of steps occurring before the main loop is no more than $c_1 + mc_2 + q(n) + c_3$, a polynomial in n.

Inside the main loop, all of the operations other than running other algorithms happen in constant time, say c_4 steps. Letter to SLP runs in constant time as well, say c_5 steps, by Lemma 2.9. In each iteration of the main loop, the algorithm Put in Lexicographic Order 2 is run on two SLPs. Each of these two SLPs was produced by either Letter to SLP or Put in Lexicographic Order 2. Those produced by Letter to SLP have size $m + 1$; by Lemma 2.55, those produced by Put in Lexicographic Order 2 have a size which is polynomial in the size of the two SLPs which were input to produce it, each of which was created in a previous iteration of the main loop. The main loop runs n times, so the size of each of the final two

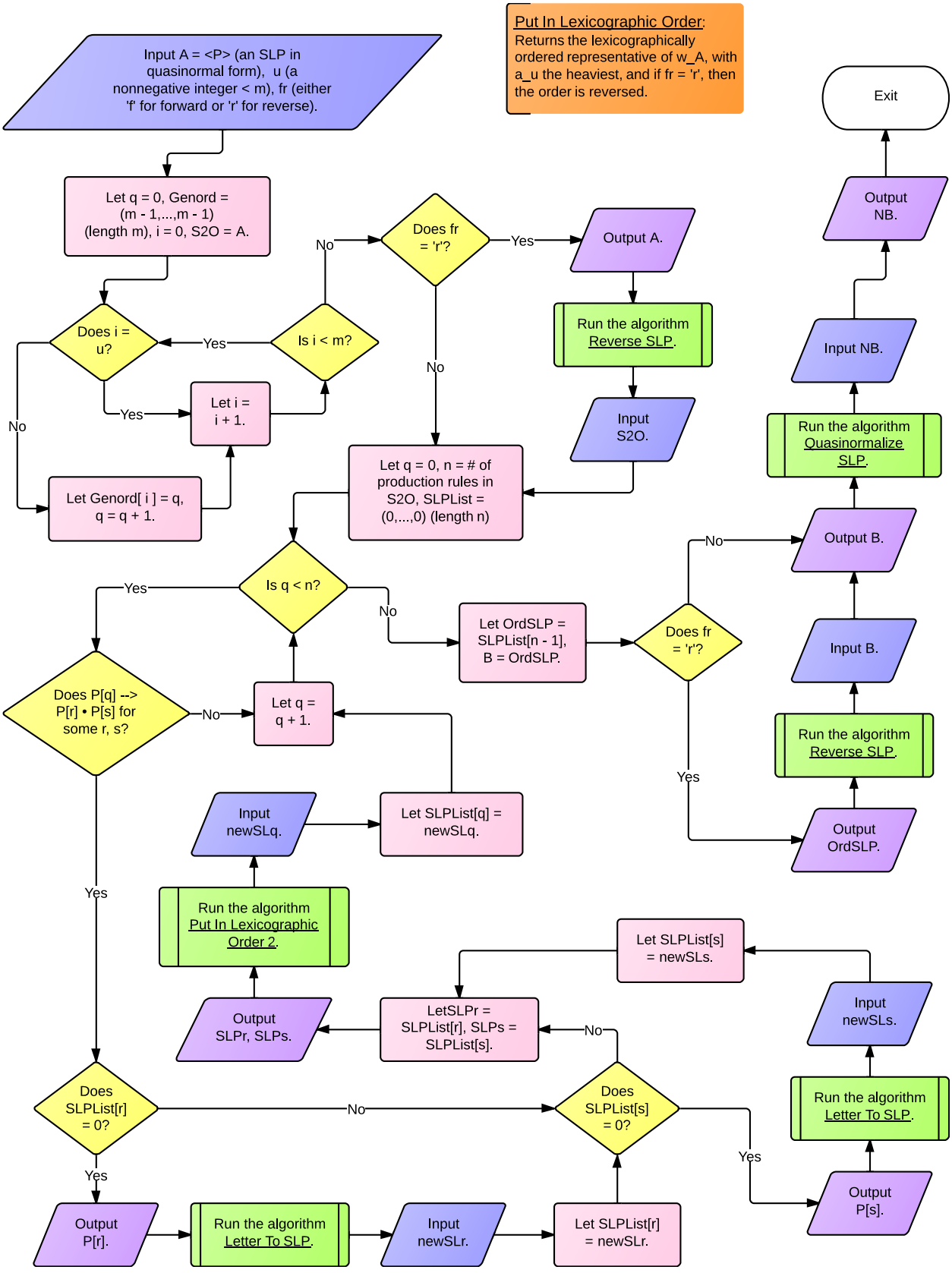


Figure 2.35: Put In Lexicographic Order

SLPs to be input into Put in Lexicographic Order 2 is bounded by a polynomial in n, say $p_1(n)$ and $p_2(n)$. Now by Lemma 2.55, Put in Lexicographic Order 2 runs in polynomial time in the sum of the sizes of the SLPs which are input, so if we call that polynomial q', then during the last iteration of the main loop, Put in Lexicographic Order 2 requires no more than $q'(p_1(n) + p_2(n))$ steps, and this is a polynomial in n. Therefore each iteration of the main loop requires no more than $c_4 + 2c_5 + q'(p_1(n) + p_2(n))$ steps; let us call this polynomial $p(n)$.

After the main loop, Reverse SLP and Quasinormalize SLP each runs in polynomial time in the size of $\mathbb{O}$rdSLP, and the size of $\mathbb{O}$rdSLP is a polynomial in $p_1(n) + p_2(n)$, and therefore a polynomial in n. Say together they require $p_3(n)$ steps. The other operations that happen after the main loop run in constant time, say c_6 steps.

Thus the number of steps required by Put in Lexicographic Order is no more than $c_1 + mc_2 + q(n) + c_3 + np(n) + p_3(n) + c_6$, which is a polynomial in n. Furthermore, the length of the SLP produced by Put in Lexicographic Order is also bounded by a polynomial in n. □

Lexico and Count, the next routine, is a key piece of the Make It Shortest 2 routine, which is the only algorithm in which it is called.

Lemma 2.58. *The algorithm Lexico and Count described by the flowchart in Figure 2.36 inputs two SLPs, $\mathbb{A}$ and $\mathbb{B}$, and an index u, and it returns two SLPs, $\mathbb{X}$Lex and $\mathbb{Y}$Lex, and two integers, C_1 and C_2, meeting the following conditions. $\mathbb{X}$Lex and $\mathbb{Y}$Lex are in quasinormal form and produce, respectively, the word w_X which is similar to w_A and in lexicographic order and the word w_Y which is similar to w_B and in reverse lexicographic order, where a_u is set to be the heaviest generator and for all $i \in \{0, 1, \ldots, u-2, u+1, ldots, m\}$, $a_i < a_{i+1}$. The values C_1 and C_2 are the number of occurrences of $a_u^{\pm 1}$ in w_X and w_Y, respectively.*

Proof. After inputting $\mathbb{A}$, $\mathbb{B}$, and u, we output $\mathbb{A}$, u, and 'f' to the algorithm Put in Lexicographic Order, which by Lemma 2.56 returns an SLP $\mathbb{X}$Lex producing the word which is

similar to w_A and in lexicographic order, with a_u given the heaviest weight. We then output $\mathbb{X}$Lex and u to Count the Occurrences, which by Lemma 2.18 returns the number C_1 of occurrences of $a_u^{\pm 1}$ in w_X, the word produced by $\mathbb{X}$Lex.

We repeat this process with $\mathbb{B}$, but with 'r' rather than 'f' so as to get reverse lexicographic order: We output $\mathbb{B}$, u, and 'r' to Put in Lexicographic Order, which returns $\mathbb{Y}$Lex, an SLP producing the word which is similar to w_B and in reverse lexicographic order, with a_u given the heaviest weight. We then output $\mathbb{Y}$Lex and u to Count the Occurrences, which returns the number C_2 of occurrences of $a_u^{\pm 1}$ in w_Y, the word produced by $\mathbb{Y}$Lex.

Finally, we return $\mathbb{X}$Lex, $\mathbb{Y}$Lex, C_1, and C_2 and exit the routine. □

Lemma 2.59. *The algorithm Lexico and Count described by the flowchart in Figure 2.36 runs in polynomial time in $n + p$, where n and p are the lengths of the SLPs which are input.*

Proof. There are no loops in this routine, just four calls to other algorithms. The first time we call Put in Lexicographic Order, by Lemma 2.57 it runs in polynomial time in n, say $P_1(n)$ steps, where n is the size of $\mathbb{A}$. The returned SLP, $\mathbb{X}$Lex, has size which is polynomial in n, say $q_1(n)$. So by Lemma 2.19, the first time we call Count the Occurrences it runs in polynomial time in $q_1(n)$, which is another polynomial in n, say $Q_1(n)$ steps. Similarly, the second time we call Put in Lexicographic Order, it runs in polynomial time in p, say $P_2(p)$ steps, where p is the size of $\mathbb{B}$. Then $\mathbb{Y}$Lex has polynomial size in p, say $q_2(p)$, so the second time we call Count the Occurrences it runs in polynomial time in $q_2(p)$, which is another polynomial in p; call it $Q_2(p)$ steps. Thus Lexico and Count runs in $P_1(n) + Q_1(n) + P_2(p) + Q_2(p)$ steps. Now $P_1(n) + Q_1(n) + P_2(p) + Q_2(p)$ is bounded by a polynomial in $n + p$, so the number of steps required to run Lexico and Count is bounded by a polynomial in $n + p$. □

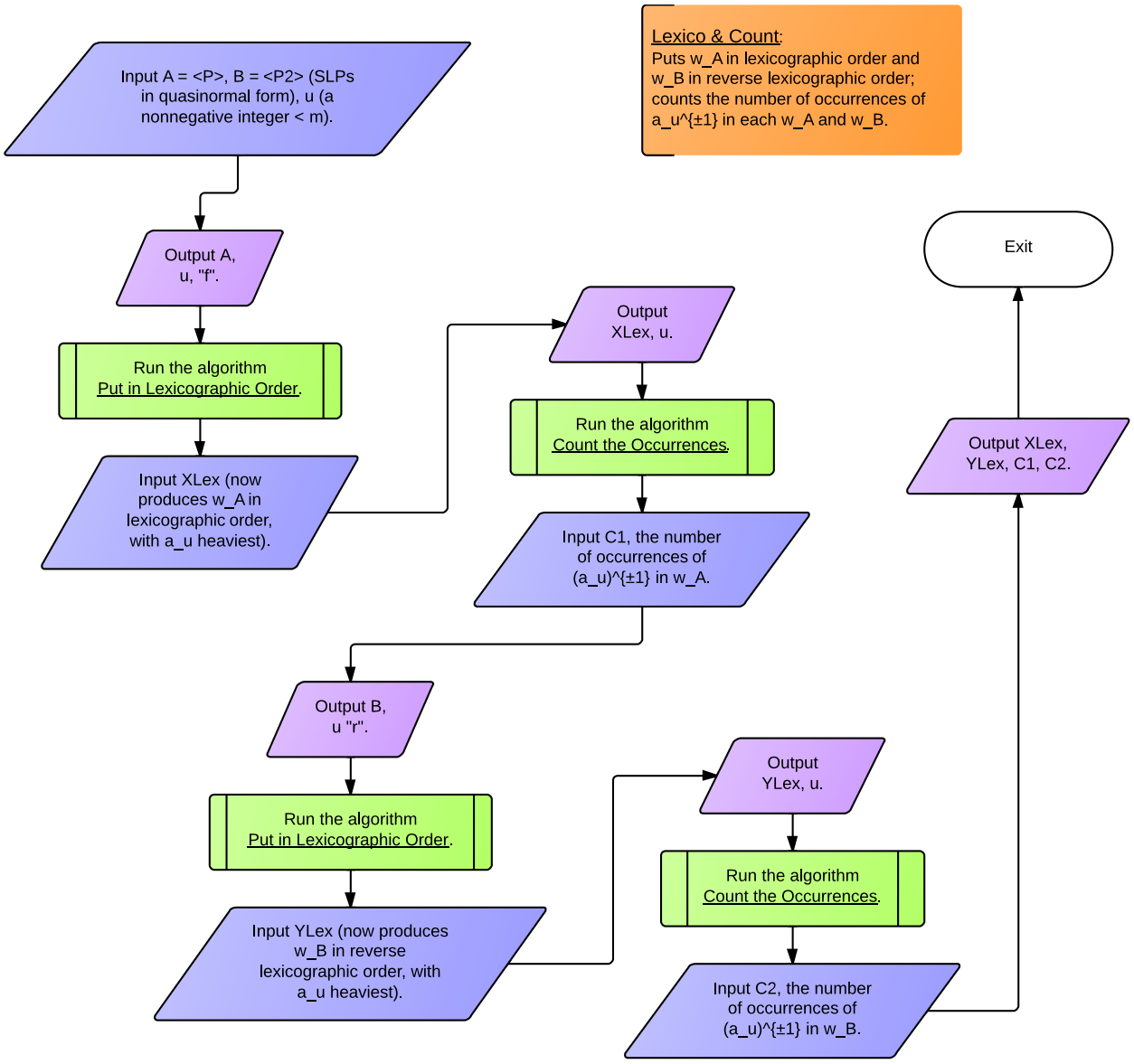


Figure 2.36: Lexico and Count

2.5 Algorithms, Part 4 – Normal Form

The remaining routines involve putting a word into its shortest form, and the very last one, ShortLex SLP, is the one to which we have been building all along. It takes an SLP and returns the SLP which produces the same group element but in shortlex form. ShortLex SLP uses, either directly or indirectly, all of the other routines discussed in this paper.

Lemma 2.60. *The algorithm Make It Shortest 2 described by the flowchart in Figure 2.37 inputs two SLPs, $\mathbb{A}$ and $\mathbb{B}$, and returns an SLP in quasinormal form which produces the word which is similar to w_A and in shortest form.*

Proof. After inputting $\mathbb{A}$ and $\mathbb{B}$, we begin by initializing the variables $\mathbb{AL}$ and $\mathbb{BL}$ to $\mathbb{A}$ and $\mathbb{B}$, respectively. In the large outer loop, we use u to step through the generators one at a time, starting with $u = 0$. Each iteration of the outer loop begins with outputting $\mathbb{AL}$, $\mathbb{BL}$, and u to the algorithm Lexico and Count, which returns two SLPs and two integers q_1 and q_2. We reassign the variables $\mathbb{AL}$ and $\mathbb{BL}$ to these returned SLPs. By Lemma 2.58, $\mathbb{AL}$ and $\mathbb{BL}$ now produce the words similar to w_{AL} and w_{BL}, respectively, and in lexicographic order with a_u given the heaviest weight, where w_{AL} and w_{BL} are the words produced by $\mathbb{AL}$ and $\mathbb{BL}$ before being reassigned to the returned SLPs. Furthermore, q_1 and q_2 are the number of occurrences of $a_u^{\pm 1}$ in w_{AL} and w_{BL}, respectively. If q_1 and q_2 are both positive, then we continue with the rest of the outer loop; otherwise we return to the beginning of the outer loop to deal with the next generator.

If $q_1 > 0$ and $q_2 > 0$, then before entering the main loop, we initialize the following variables: we set q to the minimum of $q1$ and $q2$, j to the least integer greater than or equal to $q/2$, l and r to 0, and tf to true. We want to find the occurrence of the leftmost $a_u^{\pm 1}$ in w_{AL} such that the subword beginning with that letter and ending with the last letter of w_{AL} is the inverse of the subword of w_{BL} beginning with the first letter of w_{BL} and having the same length as the subword of w_{AL}. We will set l and r so that at each step we know that fewer

than l occurrences cancel and at least r occurrence cancel, and we set j to hold the current occurrence we are testing, where we count j from the right end of w_{AL} and the left end of w_{BL}. For example, if the current value of j is 3, we are considering the 3^{rd} occurrence of a_u from the right in w_{AL}. Since q_1 is the number of occurrences of a_u in w_{AL}, the j^{th} occurrence of a_u from the right is the $(q_1 - j + 1)^{\text{th}}$ occurrence of a_u from the left. The variable tf is used to allow us to break out of the main loop at the proper time.

At the beginning of each iteration of the main loop, we output $\mathbb{AL}$, $q_1 - j + 1$, and u to Find the Occurrence. By Lemma 2.28, the number s which is returned is the position of the $(q_1 - j + 1)^{\text{th}}$ occurrence of a_u (from the left) in w_{AL}. If $s = -1$, this means there is no $(q_1 - j + 1)^{\text{th}}$ occurrence of a_u in w_{AL}, so we set tf to false in order to exit the main loop and return to the beginning of the outer loop to continue with the next generator.

If $s \neq -1$, then we output $\mathbb{AL}$ to Get Length, which by Lemma 2.4 returns the length of w_{AL}, which we assign to the variable k_1. Now $k_1 - s + 1$ is the length of the subword of w_{AL} beginning with the j^{th} occurrence of a_u from the right and ending with the rightmost letter of w_{AL}. We want to check whether or not this subword and the leftmost subword of the same length in w_{BL} are inverses, so we output $\mathbb{AL}$, $\mathbb{BL}$, and $k_1 - s + 1$ to Do They Cancel?. This routine returns a true or false value which we assign to the variable d. By Lemma 2.30, d is true if the rightmost subword of w_{AL} of length $k_1 - s + 1$ and the leftmost subword of w_{BL} of the same length are inverses and false otherwise.

If $d =$ true, then we know that at least the rightmost j occurrences of a_u in w_{AL} cancel with corresponding occurrences in w_{BL}, so we set r to j. Next we check to see if $j = l - 1$ or $r = q$. If $r = q$, then q occurrences in w_{AL} cancel with the corresponding occurrences in w_{BL}, so we do not need to check further, since q is the maximum possible number of occurrences to cancel; either w_{AL} or w_{BL} contains only q occurrences of a_u. If $j = l - 1$, then while the j^{th} occurrence from the right does cancel, the l^{th} occurrence does not, so again, we do not need to check further. Thus if either $j = l - 1$ or $r = q$, we output $\mathbb{AL}$, $\mathbb{BL}$, and s to Cancel Them,

which by Lemma 2.32 returns two SLPs, the first the producing $w_{AL}[: s-1]$ and the second one producing $w_{BL}[k_1-(s-1) :]$. In other words, the two returned SLPs produce the words w_{AL} and w_{BL}, but with the right subword of w_{AL} and the left subword of w_{BL} of length $k_1 - s + 1$ truncated. After running Cancel Them, we set tf to false so that we exit the main loop and return to the beginning of the outer loop to move on to the next generator.

Now if $d =$ true but $j \neq l-1$ and $r \neq q$, then there may be more than j occurrences of a_u which cancel, so after letting $r = j$ we let $j = r + \lceil (l-r)/2 \rceil$, putting j about halfway between the current values of r and l, and let $lasts = s$, so that if we find out during the next iteration of the main loop that no more than r occurrences of a_u cancel, we can cancel those r occurrences without recalculating the value of s for $j = r$. We then return to the beginning of the main loop.

If, on the other hand, $d =$ false, then we know that fewer than j occurrences of a_u in w_{AL} cancel with corresponding occurrences in w_{BL}. First we check to see if $j = r+1$ and $r > 0$. If so, we know that since $r > 0$ occurrences do cancel, but $r+1$ occurrences do not, we do not need to check further, but we need to cancel the $lasts^{\text{th}}$ letter and following in w_{AL}, not the s^{th} letter and following, so we set s to $lasts$. Then we output $\mathbb{AL}$, $\mathbb{BL}$, and s to Cancel Them, which returns two SLPs, the first the producing $w_{AL}[: s-1]$ and the second one producing $w_{BL}[k_1-(s-1) :]$, and we set tf to false to exit the main loop and return to the beginning of the outer loop.

If $d =$ false but either $j \neq r+1$ or $r = 0$, we next check to see if $j = 1$. If $j = 1$, then since $d =$ false, this means that the rightmost occurrence of a_u in w_{AL} does not cancel with the leftmost occurrence in w_{BL}, so no occurrences of a_u cancel. Thus we set tf to false, exit the main loop, and return to the beginning of the outer loop. If $j \neq 1$, then we know that fewer than j occurrences of a_u in w_{AL} cancel, but we still do not know exactly how many do cancel. So we set l to j and j to $r + \lceil (l-r)/2 \rceil$, putting j about halfway between the current values of r and l, and return to the beginning of the main loop.

We see that each time we exit the main loop, it is either because no occurrences of the current generator a_u in w_{AL} cancel with any occurrences in w_{BL}, or because we have run the algorithm Cancel Them to effectively cancel those occurrences. Because in each iteration of the outer loop we made a_u the heaviest generator and placed w_{AL} and w_{BL} in lexicographic and reverse lexicographic order, respectively, when we there is no occurrence of any a_i in w_{AL} which can cancel with any occurrence in w_{BL}. If after exiting the outer loop there were an occurrence of an a_i in w_{AL} and an occurrence of a_i^{-1} in w_{BL}, then during the i^{th} loop there must have been some letter a_j right of a_i in w_{AL} or left of a_i^{-1} in w_{BL} with which a_i does not commute and which does not cancel with any occurrence of a_j^{-1} in the other word: If a_j commutes with a_i, then it would not have been between a_i and a_i^{-1} (thinking of w_{AL} to the left of w_{BL}) in the i^{th} step, since a_i was made heaviest in that step. And if a_j canceled with an occurrence of a_j^{-1} in the other word, either it would have done so in the j^{th} step, in which case it would not have prevented a_i and a_i^{-1} from canceling in the i^{th} step.

Therefore, when we exit the outer loop after having considered all m generators, there are no letters in w_{AL} which can cancel with any in w_{BL}. At this point we output $\mathbb{AL}$ and $\mathbb{BL}$ to Combine SLPs, which returns $\mathbb{C}$, an SLP in quasinormal form which, by Lemma 2.20, produces the concatenation of w_{AL} and w_{BL}. Finally, we return $\mathbb{C}$ and exit the routine. □

Lemma 2.61. *The algorithm Make It Shortest 2 described by the flowchart in Figure 2.37 runs in polynomial time in $n + p$, where n and p are the lengths of the SLPs which are input. Furthermore, the length of the SLP which is returned is no more than $2^m(n + p) + 1$.*

Proof. Assigning values to variables before entering the outer loop, as well as checking to see if $u < m$, is done in constant time, say c_1 steps.

At the beginning of first iteration of the outer loop, Lexico and Count runs in polynomial time in $n + p$; each time after that, it runs in polynomial time in the sum of the sizes of $\mathbb{AL}$ and $\mathbb{BL}$, which we will see are linear in n and p, respectively. Thus in any iteration of the

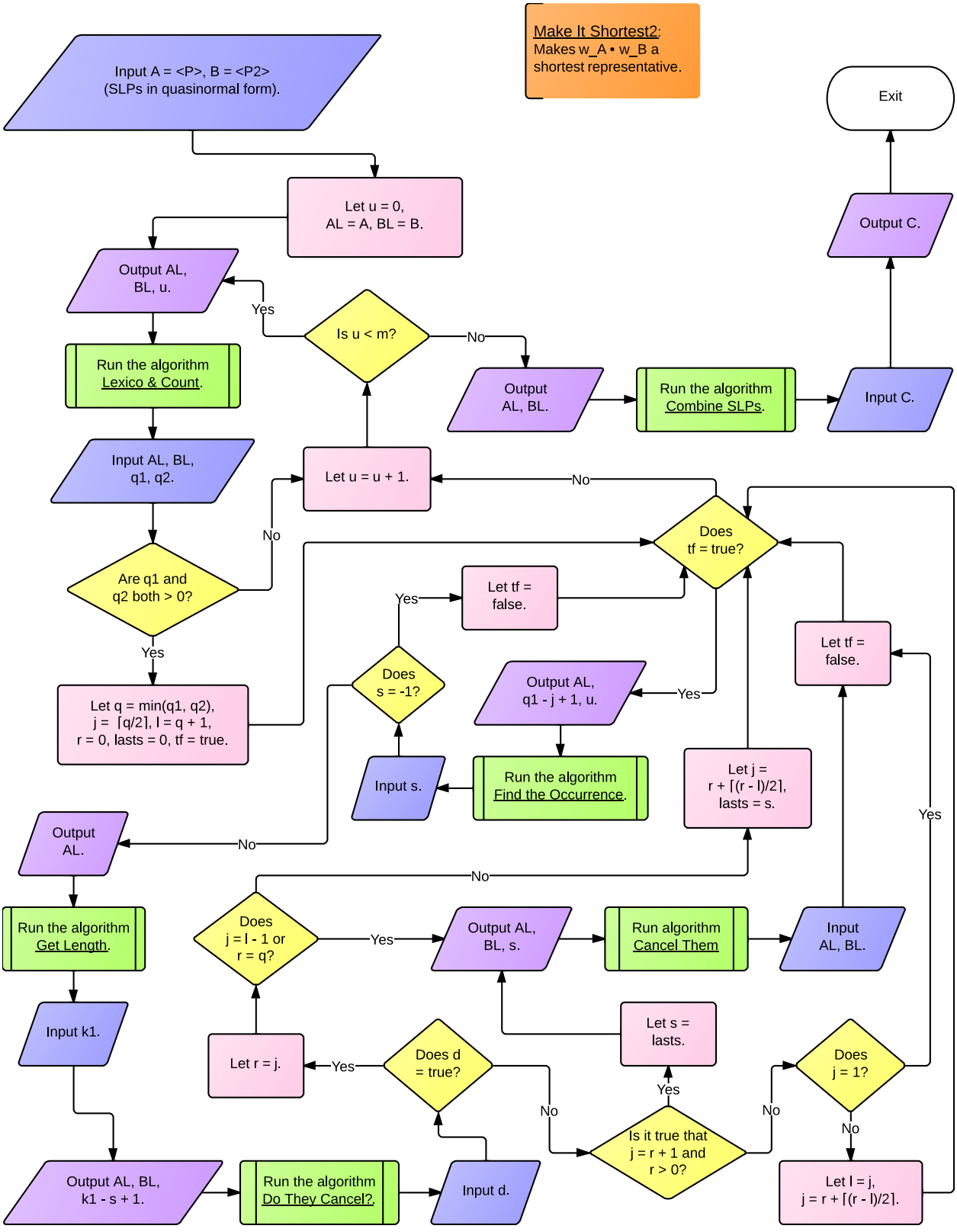


Figure 2.37: Make It Shortest 2

outer loop, the number of steps required to run Lexico and Count is bounded by a polynomial in $n+p$. Let $P_1(n+p)$ be the largest of these polynomials.

Assigning values to variables, checking to see if q_1 and q_2 are positive, and checking to see if tf is true happen in constant time, say c_2 steps. By Lemma 2.29, running Find the Occurrence happens in polynomial time in the size of $\mathbb{AL}$, which we will see is linear in n, so each time it is run during the main loop, Find the Occurrence requires no more than $P_2(n)$ steps for some polynomial P_2. Similarly, Get Length runs in polynomial time in n, say no more than $P_3(n)$ steps in any iteration of the main loop, by Lemma 2.5. By Lemma 2.31, the number of steps required by Do They Cancel? is bounded by a polynomial in the sum of the sizes of $\mathbb{AL}$ and $\mathbb{BL}$, so by a polynomial in $n+p$. Let $P_4(n+p)$ be a polynomial such that each time Do They Cancel? is run during the main loop, it runs in $P_4(n+p)$ steps or fewer. Inside the main loop, assigning values to variables and testing the values of variables all happen in constant time, say c_3 steps at most for any iteration of the main loop. The number of steps required to run Cancel Them is, by Lemma 2.33, bounded by a polynomial in the sum of the sizes of $\mathbb{AL}$ and $\mathbb{BL}$, so by a polynomial in $n+p$; say no more than $P_5(n+p)$ steps are required for running Cancel Them in any iteration of the main loop. Now, also by 2.33, the size of each of the SLPs which are output by Cancel Them is no more than double the size of the corresponding SLP which was input. Now before we get to Combine Them at the end, Cancel Them is the only algorithm run which outputs new SLPs, so since the sizes of $\mathbb{AL}$ and $\mathbb{BL}$ at most double each time Cancel Them is run, the sizes of $\mathbb{AL}$ and $\mathbb{BL}$ are at most a constant multiple of n and p, respectively, at any point in the routine before we run Combine Them.

We consider next how many times the main loop is run. For a given iteration of the outer loop, the main loop runs no more than $\log_2 q$ times, since each time through we eliminate the need to check half of the remaining occurrences about which we are uncertain. Recall that $q = \min(q_1, q_2)$, where q_1 and q_2 are the number of occurrences of a_u in w_{AL} and w_{BL}, respectively. Since each non-terminal character in an SLP has a production of at most 2,

there are no more than 2^n letters in an SLP whose length is n. So the first time through the main loop, there are no more than 2^n letters in w_{AL} and no more than 2^p letters in w_{BL}; thus $q_1 \leq 2^n$ and $q_2 \leq 2^p$, so $q \leq \min(2^n, 2^p)$. Therefore, for $u = 0$, the main loop runs no more than $\min(\log_2 2^n, \log_2 2^p) = \min(n, p)$ times. For $u = 1$, it runs no more than $\min(\log_2 2^{2n}, \log_2 2^{2p}) = \min(2n, 2p)$, and for $u = k$ in general, no more than $\min(2^k n, 2^k p)$ times. Since u ranges from 0 to $m - 1$, the number of times the main loop runs in a single iteration of the outer loop is never more than $\min(2^{m-1}n, 2^{m-1}p)$, which is bounded by $2^{m-1}(n + p)$, a linear function in $n + p$.

Thus the number of steps for a single iteration of the outer loop is bounded by $P_1(n+p)+c_2+2^{m-1}(n+p)[P_2(n)+P_3(n)+P_4(n+p)+c_3+P_5(n+p)]$, which is bounded by a polynomial in $n + p$, say $Q_1(n + p)$. The outer loop runs m times, so together all of the steps for all of the iterations of the outer loop is at most $mQ_1(n + p)$.

After exiting the outer loop, by Lemma 2.21, Combine SLPs runs in polynomial time in the sum of the sizes of the SLPs being input. As discussed above, the sizes of $\mathbb{AL}$ and $\mathbb{BL}$ when they are output to Combine SLPs are constant multiples of n and p – specifically, no more than $2^m n$ and $2^m p$, respectively. Thus the number of steps required to run Combine SLPs is bounded by a polynomial in $n + p$, say $Q_2(n + p)$. Therefore there are at most $c_1 + mQ_1(n + p) + Q_2(n + p)$ steps required to run Make It Shortest 2.

Since the sizes of $\mathbb{AL}$ and $\mathbb{BL}$ are no more than $2^m n$ and $2^m p$, respectively, when they are output to Combine SLPs, by Lemma 2.21, the size of the SLP which is output is no more than $2^m(n + p) + 1$, which is linear in $n + p$. □

Lemma 2.62. *The algorithm Make It Shortest described by the flowchart in Figure 2.38 inputs an SLP $\mathbb{A}$ and returns an SLP in quasinormal form which produces the word which is similar to w_A and in shortest form.*

Proof. After inputting $\mathbb{A}$, although it is not shown in the flowchart, we output $\mathbb{A}$ to Get

Length, and if the returned length is less than 2, we return $\mathbb{A}$ and exit the routine. Otherwise, we begin by creating a list **SLPList** of length n with each item set to 0. In the main loop, we use q to step through the production rules of $\mathbb{A}$ one at a time, starting with $q = 0$. We begin each iteration of the main loop by checking to see if the current production rule is of the form $A_q \to A_r \cdot A_s$. If not, we return to the beginning of the main loop to continue with the next production rule. If the current production rule is of the form $A_q \to A_r \cdot A_s$, we consider **SLPList**. If **SLPList**$[r] = 0$, then we have not changed its value yet, so we know that the production of A_r is a letter. Hence we output the ProdRule for A_r to Letter to SLP and set **SLPList**$[r]$ to the SLP which is returned. By Lemma 2.8, this SLP produces the letter $w(A_r)$. Similarly, if **SLPList**$[s] = 0$, we output the ProdRule for A_s to Letter to SLP and set **SLPList**$[s]$ to the SLP which is returned, which produces the letter $w(A_s)$. If **SLPList**$[r] \neq 0$, then we do not change its value; similarly we leave **SLPList**$[s]$ unchanged if its value is not 0.

Whether or not we changed the value of **SLPList**$[r]$ or **SLPList**$[s]$, we set the variables $SLPr$ and $SLPs$ to **SLPList**$[r]$ and **SLPList**$[s]$, respectively. We then output $SLPr$ and $SLPs$ to Make It Shortest 2, which by Lemma 2.60 returns an SLP producing a shortest form of the word $w(A_r) \cdot w(A_s)$. We then return to the beginning of the main loop to deal with the next production rule. Hence, at the end of each step of the main loop, **SLPList**$[q]$ contains an SLP in quasinormal form which produces a shortest form of the word $w(A_q)$. Therefore, after completing the main loop, the last entry in **SLPList**, that is, **SLPList**$[n-1]$, contains an SLP in quasinormal form which produces a shortest form of the word w_A. Thus after completing the main loop, we set $\mathbb{B}$ to **SLPList**$[n-1]$ and output $\mathbb{B}$. □

Lemma 2.63. *The algorithm Make It Shortest described by the flowchart in Figure 2.38 runs in polynomial time in n, the length of the SLP which is input. Furthermore, the length of the SLP which is returned is bounded by a linear function in n.*

Proof. The operations that happen before the first loop, as well as the time required to determine whether or not $q < n$, do not depend on n and so happen in constant time, say c_1

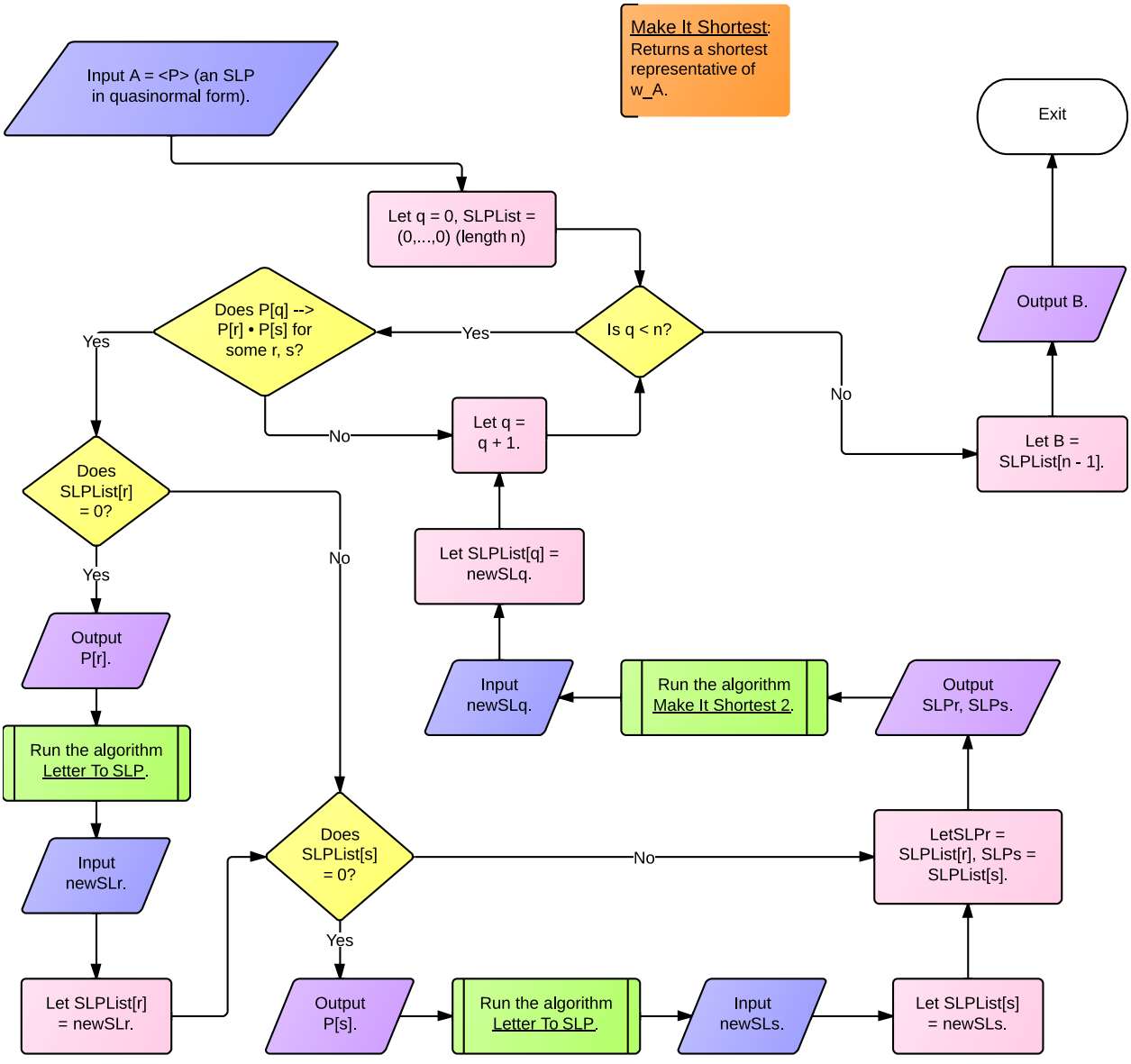


Figure 2.38: Make It Shortest

steps. Inside the main loop, all of the operations other than running other algorithms happen in constant time, say c_2 steps. Letter to SLP runs in constant time as well, say c_3 steps, by Lemma 2.9. In each iteration of the main loop, the algorithm Make It Shortest 2 is run on two SLPs. Each of these two SLPs was produced by either Letter to SLP or Make It Shortest 2. Those produced by Letter to SLP have size $m + 1$; by Lemma 2.61, those produced by Make It Shortest 2 have a size which is linear in the sum of the sizes of the two SLPs which were input to produce it, each of which was created in a previous iteration of the main loop. The main loop runs n times, so the size of each of the final two SLPs to be input into Make It Shortest 2 is bounded by a linear function in n, say $p_1(n)$ and $p_2(n)$. Now by Lemma 2.61, Make It Shortest 2 runs in polynomial time in the sum of the sizes of the SLPs which are input, so if we call that polynomial q_1, then during the last iteration of the main loop, Make It Shortest 2 requires no more than $q_1(p_1(n) + p_2(n))$ steps, and this is a polynomial in n. Therefore each iteration of the main loop requires no more than $c_2 + 2c_3 + q_1(p_1(n) + p_2(n))$ steps; let us call this polynomial $p(n)$.

The operations that happen after the main loop run in constant time, say c_4 steps. Thus the number of steps required by Make It Shortest is no more than $c_1 + np(n) + c_4$, which is a polynomial in n. Furthermore, the length of the SLP produced by Make It Shortest is bounded by a linear function in n. □

Lemma 2.64. *The algorithm Shortlex SLP described by the flowchart in Figure 2.39 inputs an SLP $\mathbb{A}$ and returns an SLP in quasinormal form which produces the word which is similar to w_A and in shortlex form.*

Proof. After inputting $\mathbb{A}$, although it is not shown in the flowchart, we check to see if $\mathbb{A}$ is in quasinormal form, and if not, we output $\mathbb{A}$ to Quasinormalize SLP and input by Lemma 2.10 an SLP which is in quasinormal form and which produces w_A. We assign the variable $\mathbb{A}$ to this new SLP. Whether or not we needed to quasinormalize $\mathbb{A}$, we continue by outputting $\mathbb{A}$ to Make It Shortest, which by Lemma 2.62 returns an SLP $\mathbb{S}$hortA in quasinormal form

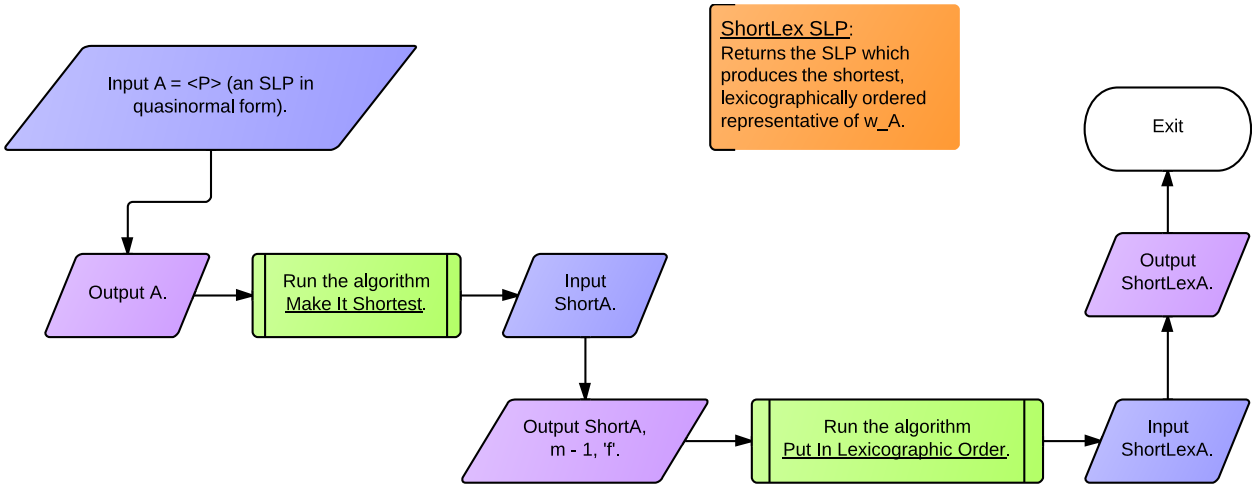


Figure 2.39: Shortlex SLP

producing a shortest form of the word w_A. We then output $\mathbb{S}$hortA, $m-1$, and 'f' to Put in Lexicographic Order, which by Lemma 2.56 returns an SLP $\mathbb{S}$hortlexA in quasinormal form which produces a lexicographically ordered word which is similar to $w(ShortA)$ and therefore to w_A. Now lexicographically ordering a word does not change its length, so $w(ShortlexA)$ is still in shortest form. Therefore the word produced by $\mathbb{S}$hortlexA is in shortlex form and is similar to w_A. We return $\mathbb{S}$hortlexA. □

Theorem 2.65. *The algorithm Shortlex SLP described by the flowchart in Figure 2.39 runs in polynomial time in n, the length of the SLP which is input.*

Proof. The algorithm Make It Shortest runs in polynomial time in n, say $p(n)$ steps, by Lemma 2.63. Furthermore, also by Lemma 2.63, the length of the SLP which it returns is linear in n, say $l(n)$. By Lemma 2.57, Put in Lexicographic Order runs in polynomial time in $l(n)$, which is also polynomial in n. Thus Shortlex SLP requires $p(n)+l(n)$ steps, which is a polynomial in n. □

CONCLUSION

We have demonstrated the existence of a polynomial-time algorithm which, given a straight line program in Chomsky normal form, provides a straight line program which produces the shortlex form of the word produced by the initial straight line program. This algorithm solves the compressed word problem for right-angled Artin groups, thereby providing a solution for the word problem for automorphism groups of RAAGs. One area of further study is developing a similar algorithm for the word problem for the outer automorphism groups of RAAGs.

APPENDIX: PYTHON CODE

The actual Python code used to create the algorithms discussed in the paper is shown below.

```
class Concat(object):
    def __init__(self, P, idx1, idx2):
        self.P = P
        self.idx1 = idx1
        self.idx2 = idx2
        self.pr1 = P[idx1]
        self.pr2 = P[idx2]

    def perform(self):
        a = self.pr1.use()
        b = self.pr2.use()
        if a == 0:
            return b
        elif b == 0:
            return a
        else:
            return a + b


class Pr1Pr(object):
    def __init__(self, P, wi):
        self.P = P
        self.wi = wi

        self.idx1 = self.wi
        self.pr1 = P[self.idx1]

    def perform(self):
        if type(self.P[self.idx1].wi) == Pr1Pr:
            a = self.P[self.idx1].wi.perform()
            return a
        else:
            a = self.pr1.use()
            return a


class ProdRule(object):
    def __init__(self, wi):
        self.wi = wi
```

```
    def use(self):
        s = self.wi
        if type(s) == Concat or type(s) == Pr1Pr:
            return s.perform()
        else:
            return s

    def LetterToSLP(self):
        QList = []
        B = SLP(QList, false)
        s = self.wi
        if type(s) == str:
            for i in range(0, m):
                if s == B.gens[i]:
                    GenIdx = i
                ProdI = ProdRule(B.gens[i])
                QList.append(ProdI)
            for i in range(0, m):
                if s == B.invgens[i]:
                    GenIdx = m + i
                ProdI = ProdRule(B.invgens[i])
                QList.append(ProdI)

            w = Pr1Pr(QList, GenIdx)
            QList.append(ProdRule(w))
            return SLP(QList, true)
        else:
            return B


class SLP(object):
    def __init__(self, P, normform):
        self.P = P
        self.normform = normform

        theGens = []
        theInvgens = []
        for i in range(m):
            aNum = 'a' + str(i)
            theGens.append(aNum)
            ANum = 'A' + str(i)
            theInvgens.append(ANum)
        self.gens = theGens
        self.invgens = theInvgens
        self.numbr = len(P)
```

```
def evaluate(self):      #returns the string produced by the SLP self
    if self.numbr == 0:
        return ''
    num = self.numbr - 1
    thisP = self.P
    finalP = thisP[num]
    aANum = finalP.use()
    if type(aANum) == Integer:
        aANum = ''
        print 'This is an empty SLP.'
    theStr = ''

    if m > 26:
        print 'This program is not currently equipped to deal with more than
            26 generators. The output will be incorrect.'
        # if m > 26, need a different approach to labeling the
        # generators & inverses

    for i in range(0, len(aANum), 2):
        nr = Integer(aANum[i+1])
        if aANum[i] == 'a':
            ch = chr(nr + 97)
        else:
            ch = chr(nr + 65)
        theStr = theStr + ch
    return theStr


def length(self):          #returns the length of the evaluated word
    if self.numbr == 0:
        return 0
    thisP = self.P
    n = self.numbr
    prL = [0 for i in range(n)] #length of word produced by each prod rule
    for i in range(n):
        if type(thisP[i].wi) == Concat:
            r = thisP[i].wi.idx1
            s = thisP[i].wi.idx2
            prL[i] = prL[r] + prL[s]
        elif type(thisP[i].wi) == Pr1Pr:
            r = thisP[i].wi.idx1
            prL[i] = prL[r]
        else:
            prL[i] = 1
    return prL[n - 1]
```

```
def FIG(self):             #returns a list indicating which generators are
    thisP = self.P          #included in the string produced by self
    n = self.numbr
    z = [0 for i in range(m)]
    if n == 0:
        return z
    for i in range(m):
        z1 = [0 for j in range(n)]
        for j in range(n):
            if type(thisP[j].wi) != Concat:
                letter = thisP[j].use()
                if letter == self.gens[i] or letter == self.invgens[i]:
                    z1[j] = 1
            else:
                thiswi = thisP[j].wi
                r = thiswi.idx1
                s = thiswi.idx2
                if z1[r] == 1 or z1[s] == 1:
                    z1[j] = 1
        if z1[n - 1] == 1:
            z[i] = 1
    return z


def quasinormalize(self): #quasinormalizes the SLP self
    thisP = self.P
    n = self.numbr
    QList = []                          #new prod rules
    LList = []                          #list of letters used
    z = [0 for i in range(n)]
    z2 = [-1 for i in range(n)]
    q = 0
    onlyLetters = true             #if no other prod rules, then make SLP
        empty
    for i in range(n):                  #put all the prod rules going to letters
        if type(thisP[i].wi) == str:    #at the beginning
            letter = thisP[i].use()
            if letter != emptyLetter:
                if letter not in LList:     #only add a letter once
                    LList.append(letter)
                    l = ProdRule(letter)
                    QList.append(l)
                    q = q + 1
                z2[i] = LList.index(letter)
    lList = []                              #don't repeat letters
    eList = []                              #list of prod rules --> ee00
    lastConcat = -1                         #position of last Concat kept
```

```
lastPr1Pr = -1                          #position of last Pr1Pr
ConcatList = [0 for i in range(n)]      #list of Concats
k = 0
for i in range(n):
    if type(thisP[i].wi) == Pr1Pr:
        w = thisP[i].wi
        r = w.wi
        z[i] = i - r + z[r]
        k = k + 1
        if r in eList:
            eList.append(i)
        else:
            onlyLetters = false
            lastPr1Pr = i
    elif type(thisP[i].wi) == str:
        letter = thisP[i].use()
        if letter == emptyLetter:
            k = k + 1
            eList.append(i)
        elif letter not in lList:       #don't increase k for new letter
            lList.append(letter)
        else:
            k = k + 1
        z[i] = i - z2[i]
    else:                               #type must be Concat
        r = thisP[i].wi.idx1
        s = thisP[i].wi.idx2
        if r in eList and s in eList:
            eList.append(i)
            z[i] = i + 1
            k = k + 1
        elif r in eList:
            z[i] = i - s + z[s]
            onlyLetters = false
            lastPr1Pr = i
            k = k + 1
        elif s in eList:
            z[i] = i - r + z[r]
            onlyLetters = false
            lastPr1Pr = i
            k = k + 1
        else:
            w = Concat(QList, r - z[r], s - z[s])
            QList.append(ProdRule(w))
            ConcatList[q] = 1
            q = q + 1
            z[i] = k
```

```
            onlyLetters = false
            lastConcat = i
    if n - 1 in eList:
        onlyLetters = true
    else:
        if lastPr1Pr > lastConcat:
            i = lastPr1Pr
            nextrule = i - z[i]
            if ConcatList[nextrule] != 0:
                QList = QList[0:nextrule + 1]
            else:                   #must have nextrule <= len(LList)
                w = Pr1Pr(QList, nextrule)
                QList.append(ProdRule(w))
    if onlyLetters:
        QList = []
    B = SLP(QList, true)
    return B


def LeftSub(self, g): #returns the SLP producing the leftmost subword
                              #of length w
    if self.numbr == 0:
        return self
    n = self.numbr
    N = n - 1
    q = 0
    QList = [ProdRule(emptyLetter) for i in range(0, 2*N + 1)]
    L = self.length()
    if g > L:
        print 'Number of letters requested is ' + str(g) + ', which is more
            than the number of letters (' + str(L) + ') in the string produced
            by this SLP.'
        return self
    elif g == L:
        return self
    elif g == 0:
        QList = []
        return SLP(QList, true)
    thisP = self.P
    prL = [0 for i in range(n)] #length of word produced by each prod rule
    while q < n:
        if type(thisP[q].wi) == Concat:
            r = thisP[q].wi.idx1
            s = thisP[q].wi.idx2
            prL[q] = prL[r] + prL[s]
            w = Concat(QList, 2*r, 2*s)
            QList[2*q] = ProdRule(w)
```

```
    elif type(thisP[q].wi) == Pr1Pr:
        r = thisP[q].wi.idx1
        prL[q] = prL[r]
        w = Pr1Pr(QList, 2*r)
        QList[2*q] = ProdRule(w)
    else:
        prL[q] = 1
        letter = thisP[q].use()
        QList[2*q] = ProdRule(letter)
    q = q + 1
q = N
h = g
used = [0 for i in range(2*N + 1)]
tf = true
while tf:
    if type(thisP[q].wi) == Concat and tf == true:
        r = thisP[q].wi.idx1
        s = thisP[q].wi.idx2
        l = prL[r]
        if l > h:
            w = Pr1Pr(QList, 2*r - 1)
            QList[2*q - 1] = ProdRule(w)
            used[2*q - 1] = 1
            used[2*r - 1] = 1
            q = r
        elif l == h:
            w = Pr1Pr(QList, 2*r)
            QList[2*q - 1] = ProdRule(w)
            used[2*q - 1] = 1
            used[2*r] = 1
            tf = false
        else:
            w = Concat(QList, 2*r, 2*s - 1)
            QList[2*q - 1] = ProdRule(w)
            used[2*q - 1] = 1
            used[2*r] = 1
            used[2*s - 1] = 1
            q = s
            h = h - l
    else:
        tf = false
hiUsed = 0
for i in range(2*N + 1):
    if used[i] == 1:
        hiUsed = i
QListNew = QList[0:hiUsed + 1]
B = SLP(QListNew, false)
```

```
    NB = B.quasinormalize()
    return NB


def RightSub(self, f):       #returns the SLP producing the rightmost
    if self.numbr == 0:           #subword of length f
        return self
    n = self.numbr
    N = n - 1
    q = 0
    QList = [ProdRule(emptyLetter) for i in range(0, 2*N + 1)]
    L = self.length()
    if f > L:
        print 'Number of letters requested is ' + str(f) + ', which is more
            than the number of letters (' + str(L) + ') in the string produced
            by this SLP.'
        return self
    elif f == L:
        return self
    elif f == 0:
        QList = []
        return SLP(QList, true)
    thisP = self.P
    prL = [0 for i in range(n)] #length of word produced by each prod rule
    while q < n:
        if type(thisP[q].wi) == Concat:
            r = thisP[q].wi.idx1
            s = thisP[q].wi.idx2
            prL[q] = prL[r] + prL[s]
            w = Concat(QList, 2*r, 2*s)
            QList[2*q] = ProdRule(w)
        elif type(thisP[q].wi) == Pr1Pr:
            r = thisP[q].wi.idx1
            prL[q] = prL[r]
            w = Pr1Pr(QList, 2*r)
            QList[2*q] = ProdRule(w)
        else:
            prL[q] = 1
            letter = thisP[q].use()
            QList[2*q] = ProdRule(letter)
            if q != 0:
                QList[2*q - 1] = ProdRule(letter)
        q = q + 1
    q = N
    h = f
    used = [0 for i in range(2*N + 1)]
    tf = true
```

```
    while tf:
        if type(thisP[q].wi) == Concat and tf == true:
            r = thisP[q].wi.idx1
            s = thisP[q].wi.idx2
            l = prL[s]
            if l > h:
                w = Pr1Pr(QList, 2*s - 1)
                QList[2*q - 1] = ProdRule(w)
                used[2*q - 1] = 1
                used[2*s - 1] = 1
                q = s
            elif l == h:
                w = Pr1Pr(QList, 2*s)
                QList[2*q - 1] = ProdRule(w)
                used[2*q - 1] = 1
                used[2*s] = 1
                q = s
                tf = false
            else:
                w = Concat(QList, 2*r - 1, 2*s)
                QList[2*q - 1] = ProdRule(w)
                used[2*q - 1] = 1
                used[2*r - 1] = 1
                used[2*s] = 1
                q = r
                h = h - l
        else:
            tf = false
    hiUsed = 0
    for i in range(2*N + 1):
        if used[i] == 1:
            hiUsed = i
    QListNew = QList[0:hiUsed + 1]
    B = SLP(QListNew, false)
    NB = B.quasinormalize()
    return NB


def BothSub(self, F, G): #returns the SLP producing the subword beginning
                             #at F & ending at G (so F = 5, G = 15 =>
    B = self.LeftSub(G)      #start with 5th letter, end with 15th, get a
    C = B.RightSub(G - F + 1) #subword of length 11)
    return C


def CountOccs(self, u): #(to count a0, put 0 in for letterIdx)
                              #Counts the number of times a letter & its
```

```
                            #inverse appear in the word produced by self
    if u >= m:
        print 'There are only ' + str(m) + ' letters in the alphabet, numbered
             0 through ' + str(m - 1) + '.'
        return 0
    GLet = self.gens[u]
    ILet = self.invgens[u]
    q = 0
    n = self.numbr
    thisP = self.P
    QList = [ProdRule(emptyLetter) for i in range(n)]
    while q < n:
        if type(thisP[q].wi) == str:
            letter = thisP[q].use()
            if letter == GLet or letter == ILet:
                QList[q] = thisP[q]
        else:
            QList[q] = thisP[q]
        q = q + 1
    B = SLP(QList, false)
    NB = B.quasinormalize()
    k = NB.length()
    return k


def Inverse(self):     #returns the SLP which produces the inverse of self
    if self.numbr == 0:
        return self
    thisP = self.P
    n = self.numbr
    q = 0
    QList = [ProdRule(emptyLetter) for i in range(0, n)]
    k = self.length()
    while q < n:
        if type(thisP[q].wi) == str:
            letter = thisP[q].use()
            if letter[0] == 'a':
                invletter = 'A' + letter[1]
            else:
                invletter = 'a' + letter[1]
            w = ProdRule(invletter)
            QList[q] = w
        elif k > 1:                #type must be Concat
            r = thisP[q].wi.idx1
            s = thisP[q].wi.idx2
            w = Concat(QList, s, r)
            QList[q] = ProdRule(w)
```

```
        else:
            QList[q] = thisP[q]
        q = q + 1
    B = SLP(QList, false)
    NB = B.quasinormalize()
    return NB


def FindTheOcc(self, p, u): #returns the position of the pth occurrence
    k = self.CountOccs(u)
    if p > k:
        j = -1
        return j
    if p == -1:
        p = k

    thisP = self.P
    n = self.numbr
    q = 0
    GLet = self.gens[u]
    ILet = self.invgens[u]
    occs = [0 for i in range(n)] #number of occurrences of u produced by each
                                 # prod rule
    prL = [0 for i in range(n)] # length of word produced by each prod rule
    while q < n:
        if type(thisP[q].wi) == Concat:
            r = thisP[q].wi.idx1
            s = thisP[q].wi.idx2
            occs[q] = occs[r] + occs[s]
            prL[q] = prL[r] + prL[s]
        elif type(thisP[q].wi) == Pr1Pr:
            r = thisP[q].wi.idx1
            occs[q] = occs[r]
            prL[q] = prL[r]
        elif type(thisP[q].wi) == str:
            prL[q] = 1
            letter = thisP[q].use()
            if letter == GLet or letter == ILet:
                occs[q] = 1
        q = q + 1

    q = n - 1
    j = 0
    tf = true
    while tf:
        if type(thisP[q].wi) == Concat and tf:
            r = thisP[q].wi.idx1
```

```
            s = thisP[q].wi.idx2
            v = occs[r]

            if v < p:
                p = p - v
                q = s
                j = j + prL[r]
            else:
                q = r
        else:
            j = j + 1
            tf = false
    return j


def RmostNoncomm(self, i): #returns the number of the gen h & its pos t
                    #of the rightmost gen not commuting with the ith gen
    y = self.FIG()
    for j in range(m):
        if y[j] == 1:
            if (i, j) in R or (j, i) in R:
                y[j] = 0
    h = -1
    t = -1
    for j in range(m):
        if y[j] == 1:
            s = self.FindTheOcc(-1, j)
            if s > t:
                h = j
                t = s
    return (h, t)


def LmostNcomList(self, iList):
    y = self.FIG()
    y2 = [0 for j in range(m)]
    L = len(iList)

    for j in range(m):
        if y[j] == 1:
            for i in range(L):
                if (iList[i], j) not in R and (j, iList[i]) not in R:
                    y2[j] = 1
                    break
    k = self.length()
    h = -1
    t = k + 1
```

```
    for j in range(m):
        if y2[j] == 1:
            s = self.FindTheOcc(1, j)
            if s < t:
                h = j
                t = s
    if h == -1:
        t = -1
    return (h, t)


def LmostComList(self, iList): #leftmost letter that commutes with
    y = self.FIG()                    #everything in iList
    y2 = y

    for j in range(m):
        if y[j] == 1:
            for i in range(len(iList)):
                if (iList[i], j) not in R and (j, iList[i]) not in R:
                    y2[j] = 0
                    break
    k = self.length()
    h = -1
    t = k + 1
    for j in range(m):
        if y2[j] == 1:
            s = self.FindTheOcc(1, j)
            if s < t:
                h = j
                t = s
    if h == -1:
        t = -1
    return (h, t)


def LmostHeavier(self, genord, i):
    y = self.FIG()
    for j in range(m):
        if y[j] == 1 and genord[j] <= genord[i]:
            y[j] = 0
    k = self.length()
    h = -1
    t = k + 1
    for j in range(m):
        if y[j] == 1:
            s = self.FindTheOcc(1, j)
            if s < t:
```

```
            h = j
            t = s
    if h == -1:
        t = -1
    return (h, t)


def RmostHeavier(self, genord, i):
    y = self.FIG()
    for j in range(m):
        if y[j] == 1 and genord[j] <= genord[i]:
            y[j] = 0
    h = -1
    t = -1
    for j in range(m):
        if y[j] == 1:
            s = self.FindTheOcc(-1, j)
            if s > t:
                h = j
                t = s
    return (h, t)


def FindFirstOccOrd(self): #returns a list of gens, ordered by position
                           #of Lmost occs
    firstocclist = []  #ith element holds position of leftmost occ of ith gen
    biggestocc = -1
    biggestoccgen = -1
    for i in range(m):
        occ = self.FindTheOcc(1, i)
        firstocclist.append(occ)
        if occ > biggestocc:
            biggestocc = occ
            biggestoccgen = i

    firstoccOrdlist = []  #the list to be returned
    didgens = [0 for i in range(m)]
    for i in range(m):
        LeftOcc = biggestocc
        LeftOccGen = biggestoccgen
        for j in range(m):
            if firstocclist[j] < LeftOcc and didgens[j] == 0:
                LeftOcc = firstocclist[j]
                LeftOccGen = j
        if LeftOcc > -1:
            firstoccOrdlist.append(LeftOccGen)
        didgens[LeftOccGen] = 1
```

```
        return firstoccOrdlist


    def Reverse(self):      #returns the SLP which produces the reverse of self
        thisP = self.P       #only used during PILO
        n = self.numbr
        q = 0
        QList = []
        while q < n:
            if type(thisP[q].wi) == str:
                QList.append(thisP[q])
            else:            #type must be Concat b/c if k <= 1,
                             # returned already in PILO
                r = thisP[q].wi.idx1
                s = thisP[q].wi.idx2
                w = Concat(QList, s, r)
                QList.append(ProdRule(w))
            q = q + 1
        B = SLP(QList, false)
        NB = B.quasinormalize()
        return NB


    def PILO(self, u, fr): #returns the SLP which produces the word produced
        # by self in lex order, or reverse lex order if fr = 'r',
        # with u the heaviest

        k = self.length()
        if k < 2:
            return self

        q = 0
        GenOrd = [m - 1 for i in range(m)] #lex ordering of gens to use
        for i in range(m):
            if i != u:
                GenOrd[i] = q
                q = q + 1
#       GenOrd[u] = m - 1              #u is last to make u heaviest

        if fr == 'r':                 #reverse the order
            SLPToOrd = self.Reverse()
        else:
            SLPToOrd = self

        #main part

        thisP = SLPToOrd.P
```

```
    n = SLPToOrd.numbr
    q = 0
    SLPList = [0 for i in range(n)]
    while q < n:
        if type(thisP[q].wi) == Concat:
            r = thisP[q].wi.idx1
            s = thisP[q].wi.idx2

            if SLPList[r] == 0:      #then r points to a generator
                SLPList[r] = thisP[r].LetterToSLP()
            if SLPList[s] == 0:
                SLPList[s] = thisP[s].LetterToSLP()

            SLPr = SLPList[r]
            SLPs = SLPList[s]
            SLPList[q] = PILO2(SLPr, SLPs, GenOrd)

        q = q + 1
    OrdSLP = SLPList[n - 1]

    #end of main part

    if fr == 'r' and OrdSLP.length() > 1: #reverse the order back
        B = OrdSLP.Reverse()
    else:
        B = OrdSLP

    NB = B.quasinormalize()
    return NB


def MakeItShortest(self): #returns the SLP which produces
                    #the word produced by self in shortest form

    if self.length() < 2:
        return self

    thisP = self.P
    n = self.numbr
    q = 0
    SLPList = [0 for i in range(n)]

    while q < n:
        if type(thisP[q].wi) == Concat:
            r = thisP[q].wi.idx1
            s = thisP[q].wi.idx2
```

```
                if SLPList[r] == 0:      #then r points to a generator
                    SLPList[r] = thisP[r].LetterToSLP()
                if SLPList[s] == 0:
                    SLPList[s] = thisP[s].LetterToSLP()

                SLPr = SLPList[r]
                SLPs = SLPList[s]
                SLPList[q] = MIS2(SLPr, SLPs)

            q = q + 1
        B = SLPList[n - 1]      #already in quasinormal form from MIS2.
        return B


    def Shortlex(self):
        A = self
        if not self.normform:
            A = self.quasinormalize()

        ShortSLP = A.MakeItShortest()
        ShortlexSLP = ShortSLP.PILO(m - 1, 'f')
        return ShortlexSLP


def StrToSLP(str2use):
    QList = []
    TheSLP = SLP(QList, false)

    L = len(str2use)
    if L == 0:
        return TheSLP

    q = 0
    for i in range(0, m):
        ProdI = ProdRule(TheSLP.gens[i])
        QList.append(ProdI)
        q = q + 1
    for i in range(0, m):
        ProdI = ProdRule(TheSLP.invgens[i])
        QList.append(ProdI)
        q = q + 1

    QGenIdx = [-1 for i in range(L)]
    for i in range(L):
        nr = ord(str2use[i])
        if nr >= 97:
            genNum = nr - 97
```

```
            QGenIdx[i] = genNum
        else:
            invgenNum = nr - 65
            QGenIdx[i] = m + invgenNum

    if L > 2:
        numLevels = ceil(log(L,2))
    else:
        numLevels = 1
    QIdx = []
    for j in range(1, numLevels + 1):
        if L % 2 != 0:                      #L is odd
            w = Pr1Pr(QList, QGenIdx[0])
            startnum = 1
        else:
            w = Concat(QList, QGenIdx[0], QGenIdx[1])
            startnum = 2
        QList.append(ProdRule(w))
        QIdx.append(q)
        q = q + 1

        for i in range(startnum, L, 2):
            w = Concat(QList, QGenIdx[i], QGenIdx[i + 1])
            QList.append(ProdRule(w))
            QIdx.append(q)
            q = q + 1
        L = ceil(L/2)
        QGenIdx = QIdx
        QIdx = []
    TheSLP = SLP(QList, false)
    NormTheSLP = TheSLP.quasinormalize()
    return NormTheSLP


def EmptyLetterSLP():
    QList = []
    B = SLP(QList, false)
    q = 0
    for i in range(0, m):
        ProdI = ProdRule(B.gens[i])
        QList.append(ProdI)
        q = q + 1
    for i in range(0, m):
        ProdI = ProdRule(B.invgens[i])
        QList.append(ProdI)
        q = q + 1
    w = ProdRule(emptyLetter)
```

```
    QList.append(w)
    w = Pr1Pr(QList, q)
    QList.append(ProdRule(w))
    C = SLP(QList, false)
    return C


def CombineSLPs(A, B): #returns the SLP which produces the concatenation
                                #of the words produced by A & B
    if A.length() == 0:
        return B
    elif B.length() == 0:
        return A

    n = A.numbr
    p = B.numbr
    QList = [ProdRule(emptyLetter) for i in range(0, n + p + 1)]
    for q in range(n):
        QList[q] = A.P[q]
    P2 = B.P
    for q in range(p):
        if type(P2[q].wi) == str:
            letter = P2[q].use()
            QList[n + q] = ProdRule(letter)
        elif type(P2[q].wi) == Pr1Pr:
            r = P2[q].wi.idx1
            w = Pr1Pr(QList, r + n)
            QList[n + q] = ProdRule(w)
        else:                   #must be Concat, since we quasinormalized it
            r = P2[q].wi.idx1
            s = P2[q].wi.idx2
            w = Concat(QList, r + n, s + n)
            QList[n + q] = ProdRule(w)
    w = Concat(QList, n - 1, p + n - 1)
    QList[n + p] = ProdRule(w)
    C = SLP(QList, false)
    NC = C.quasinormalize()
    return NC


def DoTheyCancel(A, B, r): #determines if the last r letters in A
    k1 = A.length()           #& the first r letters in B form inverse
    k2 = B.length()           #words
    if r > k1:
        return false
    elif r > k2:
        return false
```

```
    C = A.RightSub(r)
    D = B.LeftSub(r)
    E = D.Inverse()
    if C.evaluate() == E.evaluate(): #this is where Plandowski's
        return true                              #algorithm would be used
    else:
        return false


def CancelThem(A, B, s): #cuts sth letter and following from A &
    QList = []              # corresponding subword from B
    k1 = A.length()
    k2 = B.length()
    k = k2 - k1 + s - 1   #k = k2 - (k1 - (s - 1)); k1 - (s - 1) is length of
        string
                         # cut from A; want to cut same from B
    if s > k1:
        print 'There are not ' + str(s) + ' letters in the first SLP; cannot
            cancel.'
        return (A, B)
    elif k > k2:
        print 'There are not ' + str(k) + ' letters in the second SLP; cannot
            cancel.'
        return (A, B)
    D = A.LeftSub(s - 1)
    E = B.RightSub(k)
    ND = D.quasinormalize()
    NE = E.quasinormalize()
    return (ND, NE)


def PILO2(SLP1, SLP2, genord): #returns the SLP which puts the concatenation
          #of the words produced by SLP1 & SLP2 in lex order, where genord gives
              the
          #ordering of the generators, assuming SLP1 and SLP2 already produce reps
                                    #in that lex order
    tf = true
    while tf:
        fOoL = SLP2.FindFirstOccOrd()
        SLP1Mod = SLP1
        SLP2Mod = SLP2
        MoveOccurred = false

        m2 = len(fOoL)
        for i in range(m2):
            c = fOoL[i]
            CanMove = true
```

```
numcs = SLP2Mod.CountOccs(c)
while numcs > 0:
    cpos = SLP2Mod.FindTheOcc(1, c)

    #make sure c commutes with all letters to its left in SLP2Mod
    Lofc = SLP2Mod.LeftSub(cpos - 1)
    LofcGens = Lofc.FIG()
    for j in range(m):
        if LofcGens[j] == 1:
            if (c, j) not in R and (j, c) not in R:
                CanMove = false
                break                   #breaks out of for loop

    if CanMove:
        (b, bpos) = SLP1Mod.RmostNoncomm(c)
        l1 = SLP1Mod.length()
        if bpos != -1:
            Rofb = SLP1Mod.RightSub(l1 - bpos)
        else:
            Rofb = SLP1Mod
        (z, zpos) = Rofb.LmostHeavier(genord, c)
        if zpos == -1:
            CanMove = false
            break
        if bpos != -1:
            realzpos = zpos + bpos      #pos in SLP1Mod, not Rofb
        else:
            realzpos = zpos
        zRofz = SLP1Mod.RightSub(l1 - realzpos + 1)
        zRofzGens = zRofz.FIG()

        #want list of all gens in LofcGens & zRofzGens
        genList = []
        for j in range(m):
            if LofcGens[j] ==1 or zRofzGens[j] == 1:
                genList.append(j)

        l2 = SLP2Mod.length()
        cRofc = SLP2Mod.RightSub(l2 - cpos + 1)
        (h, hpos) = cRofc.LmostNcomList(genList)
        if hpos > 1:
            Lofh = cRofc.LeftSub(hpos - 1)
            (e, epos) = Lofh.LmostHeavier(genord, z)
        else:
            Lofh = cRofc
            (e, epos) = Lofh.LmostHeavier(genord, z)
```

```
if epos == -1:
    gpos = 1
else:
    gpos = epos - 1

#see if anything in the c-g block moves left of b
cgBlock = Lofh.LeftSub(gpos)
#len(cgBlock.evaluate()) is gpos
if gpos > 1 and bpos != -1:
    RofbGens = Rofb.FIG()
    (f, fpos) = cgBlock.LmostComList([b])
    while fpos != -1 and f != -1:
        cf1Block = cgBlock.LeftSub(fpos - 1)
        cf1Gens = cf1Block.FIG()
        checkGenList = []
        for j in range(m):
            if RofbGens[j] == 1 or cf1Gens[j] == 1 or j == b:
                checkGenList.append(j)

        fgBlock = cgBlock.RightSub(gpos - fpos + 1)
        (newf, newfpos) = fgBlock.LmostComList(checkGenList)
        if newfpos == -1:
            fpos = -1
        elif newfpos == 1: #then f comm w/all in checkGenList
            #check to see if something heavier than f left of b
            # (or b)
            fMovesLofb = true
            bLofb = SLP1.LeftSub(bpos)
            (d, dpos) = bLofb.RmostHeavier(genord, f)
            if dpos == -1:
                fMovesLofb = false
            else:
                #len(bLofb.evaluate()) is bpos
                dbBlock = bLofb.RightSub(bpos - dpos + 1)
                dbGens = dbBlock.FIG()
                for j in range(m):
                    if dbGens[j] == 1 and (j, f) not in R and (f,
                         j) not in R:
                        fMovesLofb = false
                        break

            if fMovesLofb:
                gpos = fpos - 1  #f doesn't move with c now
                fpos = -1
            else:
                checkGenList.append(f)
                f1gBlock = cgBlock.RightSub(gpos - fpos)
```

```
                                (f2, f2pos) = f1gBlock.LmostComList(checkGenList)
                                f = f2
                                fpos = f2pos + fpos
                            else:
                                f = newf
                                fpos = newfpos + fpos - 1

                        #move the c-g block just left of z & move the boundary between
                                                    #SLP1Mod & SLP2Mod
                        cgBlock = Lofh.LeftSub(gpos)
                        Lofz = SLP1Mod.LeftSub(realzpos - 1)
                        SLP1ModLeft = CombineSLPs(Lofz, cgBlock)
                        SLP1ModRight = CombineSLPs(zRofz, Lofc)
                        SLP1Mod = CombineSLPs(SLP1ModLeft, SLP1ModRight)

                        realgpos = gpos + cpos - 1     #pos in SLP2Mod, not Lofh
                        MoveOccurred = true
                        Rofg = SLP2Mod.RightSub(l2 - realgpos)
                        SLP2Mod = Rofg
                        numcs = SLP2Mod.CountOccs(c)
                    else:
                        break #go to next generator; no more of this one can move now
            if MoveOccurred:
                SLP1 = SLP1Mod
                SLP2 = SLP2Mod
            else:
                tf = false      #nothing could move; finished
        C = CombineSLPs(SLP1Mod, SLP2Mod)
        return C


def LexicoAndCount(A, B, u): #Returns A in lex order w/uth generator
        AL = A.PILO(u, 'f')     #heavier, B in reverse lex order w/uth gen
        Count1 = A.CountOccs(u) #heavier, # of occurrences of uth gen in A
        BL = B.PILO(u, 'r')     #& B
        Count2 = B.CountOccs(u)
        return (AL, BL, Count1, Count2)


def MIS2(A, B):  #Returns the SLP producing a shortest rep
                 #of the concatenation of the words produced by A & B.
                 #Assumes A and B already produce shortest reps.
        AL = A
        BL = B

        for u in range(m):
            (AL, BL, q1, q2) = LexicoAndCount(AL, BL, u)
```

```
        if q1 > 0 and q2 > 0:
            q = min(q1, q2)
            j = ceil(q/2)
            l = q + 1
            r = 0
            lasts = 0
            tf = true
            while tf:
                s = AL.FindTheOcc(q1 - j + 1, u)
                if s == -1:
                    tf = false
                else:
                    k1 = AL.length()
                    d = DoTheyCancel(AL, BL, k1 - s + 1)
                    if d:
                        r = j
                        if j == l - 1 or r == q:
                            (AL, BL) = CancelThem(AL, BL, s)
                            tf = false
                        else:
                            j = r + ceil((l - r)/2)
                            lasts = s
                    else:
                        if j == r + 1 and r > 0:
                            (AL, BL) = CancelThem(AL, BL, lasts)
                            tf = false
                        elif j == 1:
                            tf = false       #none cancel
                        else:
                            l = j
                            j = r + ceil((l - r)/2)
    C = CombineSLPs(AL, BL)
    return C


#Global Variables
#------------------

emptyLetter = 'ee00'

R = [(0,1), (2,3), (1,4), (3,0), (0,2), (1,2)]   #list of commutators;
                                    # should never contain (i,i) for any i

m = 5                #number of generators
```

BIBLIOGRAPHY

[1] I. Agol. The virtual Haken conjecture. arXiv: 1204.2810 [math.GT].

[2] R. Charney. An introduction to right-angled Artin groups. *Geometriae Dedicata*, 125:141–158, 2007.

[3] F. Haglund and D. Wise. Special cube complexes. *Geom. Funct. Anal.*, 17(5):1551–1620, 2008.

[4] S. Hermiller and J. Meier. Algorithms and geometry for graph products of groups. *J. Algebra*, 171(1):230–257, 1995.

[5] M. Laurence. A generating set for the automorphism group of a graph group. *Journal of the London Mathematical Society*, Second Series 52(2):318–334, 1995.

[6] M. Lohrey and S. Schleimer. Efficient computation in groups via compression. In *Computer Science – Theory and Applications*.

[7] W. Plandowski. Testing equivalence of morphisms on context-free languages. In *Algorithms – ESA '94*.

[8] S. Schleimer. Polynomial-time word problems. *Commentarii Mathematici Helvetici*, 83(4):741–765, 2008.

[9] H. Servatius. Automorphisms of graph groups. *Journal of Algebra*, 126(1):34–60, 1989.

[10] L. VanWyk. Graph groups are biautomatic. *J. Pure Appl. Algebra*, 94(3):341–352, 1994.