

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

**Experiments in Group Theory:
Group-Theoretic Algorithms in One-Relator Groups**

By

Marvin Shumowitz

A dissertation submitted to the Graduate Faculty in Mathematics in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

1996

UMI Number: 9618103

**Copyright 1996 by
Shumowitz, Marvin**

All rights reserved.

**UMI Microform 9618103
Copyright 1996, by UMI Company. All rights reserved.**

**This microform edition is protected against unauthorized
copying under Title 17, United States Code.**

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

©1996

Marvin Shumowitz

All Rights Reserved

This manuscript has been read and accepted by the Graduate Faculty in Mathematics in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

January 30, 1996
Date

Michael Anshel
Chair of Examining Committee

January 30, 1996
Date

Isaac Chavel
Executive Officer

M. Anshel

B. Randol

A. Vasquez

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Acknowledgments

This thesis could not have been completed without the help and encouragement of my advisor, Prof. Michael Anshel. Additionally, I thank Prof. Burton Randol for additional help in this paper.

Abstract

Experiments in Group Theory: Group-Theoretic Algorithms in One-Relator Groups

by

Marvin Shumowitz

Advisor: Michael Anshel

This dissertation improves on two algorithms in the theory of one-relator groups. The first of these is the algorithm given by Abe Shenitzer to determine if we can represent a one-relator group as the free product of two of its subgroups. Using the natural measure of complexity, the number of rewritings required, this paper gives a linear algorithm as opposed to the exponential one given by Shenitzer. The second of these algorithms involves the determination of whether a one relator group has a trivial center. This expands the work done by Gilbert Baumslag and Tekla Taylor, and complements the work of Frank Levin. In looking for groups with a trivial center, one only needs to look at two-generator groups, $G = \langle a, b : R = I \rangle$. The Baumslag-Taylor algorithm divides the class of one-relator groups into two subclasses, those with finite and infinitely generated normal closures of b in G . What they show is that if the group is in the later subclass and the number of b 's is > 2 , then it has a trivial center. In the process, efficient pattern matching algorithms were used in improving these group-theoretic algorithms. An inference from the trivial center program is that all groups with a trivial center are those in which the normal subgroup generated by one of the variables is not finitely generated.

Table of Contents

vi

Chapter 1.

Introduction.	Pg. 1.
Equivalent Sets of Elements in a Free group.....	Pg. 4
Shenitzer's Algorithm for Decomposability of One-Relator Groups. . . .	Pg. 8
Baumslag-Taylor Algorithm to Determine if a One-Relator Group has a Trivial Center	Pg.11

Chapter 2.

Positive One-Relator Groups.	Pg. 14
Case Where Number of Generators is greater than 3 and $l(R)$ is Odd...	Pg. 20
Results with Strings of Even Length with 3 Generators.	Pg. 21
Case Where number of Generators is Greater than 3 of all lengths.. . . .	Pg. 22
Statistics for Decomposable 3-Generator One-Relator Groups	Pg. 24
Turing Machine Representation of Shenitzer's Algorithm.. . . .	Pg. 25
Application to the Thue-Morse Sequence.	Pg. 30

Chapter 3.

An Improved Algorithm for Determining if a One- Relator group has a Trivial Center.. . . .	Pg. 31
---	--------

Chapter 4.

Pattern Matching with an Ordered Set of Keywords.. . . .	Pg. 35
--	--------

Appendices.	Pg. 41
---------------------	--------

Bibliography	Pg. 97
------------------------	--------

Chapter 1

Introduction

Let G be a group, let X be a subset of G , and let X^{-1} be the set of inverses of the elements of X .

Definition 1 X generates G if every element of G can be expressed as a product of the elements of

$$X \cup X^{-1}.$$

Definition 2 A finite string of elements of $X \cup X^{-1}$ is a word. If the word takes the value 1 then it is termed a relator. We write

$$G = \langle X, R \rangle$$

if G is generated by X , and R is a defining set of relators.

Definition 3 If both X and R are finite, then G is termed a finite presentation.

Combinatorial group theory is the branch of group theory that looks at groups in terms of their generators and relators. One of the most

studied areas is the field of one-relator groups. This paper demonstrates improved algorithms to determine if a one-relator group can be expressed as a free product of two of its non-trivial subgroups and if it has a trivial center.

One-relator groups have been much investigated in recent years. One of the reasons is that they are a natural extension of free groups. The basic theorems about these groups are the Freiheitssatz and the solvability of the word problem, proved by Wilhelm Magnus in the thirties.

Theorem 4 (Magnus) *The Freiheitssatz[1]-Let $G = \langle t, b, c, \dots; R \rangle$ where R is cyclically reduced. If L is a subset of $\{t, b, c, \dots\}$ which omits a generator occurring in R , then the subgroup M generated by L is freely generated by L .*

A proof of this theorem involves exhibiting G as an HNN extension using the automorphisms $t \rightarrow t, t^i b t^{-i} \rightarrow b_i, t^i c t^{-i} \rightarrow c_i, \dots$, where the exponent sum of a generator t occurring in R is zero, and defining the subgroup $H = \langle b_\mu, \dots, b_\lambda, c_i, d_i, \dots; i \in \mathbb{Z}, S \rangle$ where μ and λ are the minimum and maximum subscripts occurring in S (the result of R after the automorphism). In the case where there is no generator with exponent sum zero, then the automorphism $t \rightarrow y x^{-\beta}, b \rightarrow x^\alpha, c \rightarrow c, d \rightarrow d, \dots$, where α and β are the exponent sum of t and β in R , is used.

As can be seen, the idea of using automorphisms to prove results in one-relator groups goes back to the Freiheitssatz. Following this method, Abe Shenitzer[2] gave a method to determine whether a one-relator group can be decomposed into a free product of two of its subgroups. Automorphisms were also an important part of the Baumslag-Taylor algorithm[7] to determine the center.

Equivalent sets of elements in a free group

We briefly review the results of J. H. C. Whitehead[3] in 1936 to show the theoretical background for Shenitzer's algorithm. Let us first begin with some basic definitions:

Definition 5 *The length of an element of a group is the sum of the exponents on the generators in the element.*

Definition 6 *A set of generators $a = (a_1, a_2, a_3, \dots, a_n)$ of a group G is said to be minimal if*

$$l(a) = \sum_{\lambda=1}^m l(a_\lambda)$$

(where $l(a_\lambda)$ represents the length of the generator a_λ) is a minimum among all sets of generators for G .

Definition 7 *By a T-transformation on the generators $\alpha_1, \dots, \alpha_n$ of the free group F is meant a mapping of the form:*

$$T_{\alpha_k} = \alpha_k \text{ for some fixed } k, i \leq k \leq n,$$

$$T_{\alpha_i} = \alpha_k \text{ or } \alpha_i \alpha_k^\varepsilon \text{ or } \alpha_k^\varepsilon \alpha_i \text{ or } \alpha_k^\varepsilon \alpha_i \alpha_k^{-\varepsilon}, i \neq k, i \leq k \leq n. (\varepsilon = \pm 1)$$

The α_k is referred to as the distinguished or stable symbol.

Definition 8 *A level transformation is a mapping that preserves the length of the word or generator that it is applied to.*

Example 9 *Let $G = \langle a, b, c : R \rangle$. Let abc be an element of the group G . Let us take our stable symbol to be a . The resulting word resulting from the following transformation:*

$$T_a = a$$

$$T_b = a^{-1}b$$

$$T_c = ca$$

will be $aa^{-1}bca = bca$. So the above transformation on abc is level, since abc and bca have the same length.

Theorem 10 (Whitehead) *Any two minimal sets of generators of G are interchangeable by level T -transformations.*

Proof. Let (α) and (β) be any two minimal reduced sets of generators for G . They both contain the same number of elements, m , and assuming them to be ordered so that

$$l(\alpha_\lambda) \leq l(\alpha_\mu)$$

$$l(\beta_\lambda) \leq l(\beta_\mu)$$

if $\lambda < \mu$, we have

$$l(\alpha_\lambda) = l(\beta_\lambda) = k_\lambda$$

$(\lambda = 1, \dots, m)$.

If $m = 0$, the theorem is obviously true. The case for $m > 0$ will be proved by induction on m .

Let k_s be the first of $k_1, \dots, k_m$ which is the same as k_m . If $s > 1$ and $\lambda < s \leq \rho$ we have $l(\beta_\lambda) < l(\alpha_\rho)$, $l(\alpha_\lambda) < l(\beta_\rho)$ and it follows that $(\beta_1, \dots, \beta_{s-1})$ depend only on $(\alpha_1, \dots, \alpha_{s-1})$ and vice versa. Any subset of a minimal set is minimal and if $s > 1$ it follows from the hypothesis of the induction that $(\beta_1, \dots, \beta_{s-1}) \rightarrow (\alpha_1, \dots, \alpha_{s-1})$ by a sequence of level transformations. Also it follows that

$$(\alpha_1, \dots, \alpha_{s-1}, \beta_s, \dots, \beta_m) \rightarrow (\alpha_1, \dots, \alpha_{s-1}, \gamma_s, \dots, \gamma_m)$$

by level transformations, where

$$\gamma_\lambda = \alpha_{\rho_\lambda}^\varepsilon \text{ or } \alpha_{\rho_\lambda}^{-1} \alpha_{\sigma_\lambda}$$

where $s \leq \lambda, \rho_\lambda, \sigma_\lambda \leq m$.

If $s = 1$ it follows from the same argument that $(\beta) = (\gamma)$ is related to (α) in this way.

If each γ is an α^ε , a further level transformation $(\gamma) \rightarrow (\alpha_s, \dots, \alpha_m)$ carries $(\alpha_1, \dots, \alpha_{s-1}, \gamma_s, \dots, \gamma_m)$ into (α) . Otherwise we can arrange the notation so that

$$\gamma_\lambda = \alpha_\lambda$$

$$\gamma_\mu = \alpha_{\rho_\mu}^{-1} \alpha_{\sigma_\mu}$$

where $\lambda = s, \dots, t$ and $\mu = t + 1, \dots, m$, and $t = s - 1$ if no γ is an α^ε . If $\rho_\mu > t$, $\sigma_\mu > t$ for every value of μ the determinant of the linear transformation

$$\begin{aligned}\gamma_\lambda &= \alpha_\lambda \\ \gamma_\mu &= -\alpha_{\rho_\mu} + \alpha_{\sigma_\mu}\end{aligned}$$

calculated to the modulus 2 would be zero. Therefore some γ is an α , and $\rho_\mu \leq t$ or $\sigma_\mu \leq t$ for at least one value of μ , say $\gamma_{t+1} = \alpha_t^{-1} \alpha_\sigma$. Then $\sigma > t$, or the set (γ) would not be independent, and we may suppose $\sigma = t + 1$. Applying the level transformation

$$\gamma_{t+1}^{-1} = \gamma_i \gamma_{i+1} = \alpha_{i+1}$$

to (γ) , the theorem follows from a second induction on $m - t$. ■

Shenitzer's Algorithm for Decomposability of one-relator groups.

Abe Shenitzer in 1955 proved that there is an algorithm to determine if a one-relator group can be expressed as the free product of two of its proper subgroups. This theorem used the above results of Whitehead in order to select the transformations. This problem was suggested to Shenitzer by his advisor, Wilhelm Magnus, when he was a student at New York University. In [4], their unique relationship is discussed, as is also Magnus's intertwining of mathematics and philosophy.

Definition 11 *A one-relator group is decomposable if it can be expressed as the free product of two of its non-trivial subgroups. A decomposable string is the relator of a decomposable one-relator group.*

Let G be a group with generators $a_v, v = 1, 2, \dots, n$. Any automorphism A of the group maps G to an isomorphic group G' . G' can be defined by transcribing the original relations in terms of A_{a_v} . Grushko's Theorem[5] implies that at least the decomposability of G into a free product of two of its proper subgroups can be made obvious by finding an appropriate automorphism. In the case of one-relator groups, we need

to find an automorphism that when applied to the relator results in a generator of G not being involved in a relator of G' . In Shenitzer's algorithm, the automorphisms used were T - transformations.

In order to justify the algorithm, the following theorems are needed:

Theorem 12 (Shenitzer) *For a product of disjoint minimal words to be minimal, it is necessary and sufficient that each factor W_p of the product be non-trivial. (i.e. $L(W_p) \geq 1$ for all W_p).*

Theorem 13 (Shenitzer) *Let the absolute value of all exponents in $W(a_1, \dots, a_n)$ be greater than or equal to 2. Then W is minimal.*

Theorem 14 (Shenitzer) *Let $W = W(a_1, \dots, a_n)$. Let W_1 and W_2 be two minimal forms of W . Then W_1 and W_2 contain the same number of distinct symbols.*

Theorem 15 (Grushko)[5] *A one-relator group with $n \geq 2$ generators and relator involving the n generators is decomposable if and only if it is possible to reduce the number of distinct generators by applying automorphisms to the generators.*

By applying the above two theorems, we can prove the following:

Theorem 16 (Shenitzer) *Let $G = \langle a_1, \dots, a_n; R(a_1, \dots, a_n) = 1 \rangle$ where all a_i are involved in R . Let H be the free product of an infinite cyclic*

group $\langle a \rangle$ and a nontrivial group B with generators $b_\gamma \neq a$. Then G is isomorphic to H if and only if any minimal form of R contains at most $n - 1$ distinct a'_i s.

Proof. Sufficiency is obvious. To prove necessity, we assume that $G \simeq H$ and that some minimal form of R has n distinct symbols then so does every minimal form. By Grushko's Theorem, it is possible to find a representation of G such that the relator has at most $n - 1$ distinct symbols. Minimizing this word, we obtain a minimal form of R containing at most $n - 1$ symbols. This leads to a contradiction to Theorem 8. ■

Two more of Shenitzer's proofs which are useful are

Theorem 17 (Shenitzer) *Let $K = \langle a_1, a_2; R(a_1, a_2) \rangle$, $G = \langle a, b; b^n = 1 \rangle$ then for G to be isomorphic to K , it is necessary and sufficient that R be cyclically equivalent to $\alpha^n(a_1, a_2)$, where $\alpha(a_1, a_2)$ is a primitive element of the free group $F(a_1, a_2)$.*

Theorem 18 (Shenitzer) *Let $G = \langle X : R \rangle$ where $R = V^m$ and V minimal then G is not a decomposable group.*

Baumslag-Taylor Algorithm to Determine if a One-Relator Group has a Trivial Center.

Murasugi[6] showed that if a non-abelian one-relator group G has a non-trivial center, then Z , the center of G , is infinite cyclic. It has also been proven that no algorithm to determine the center for all finitely presented groups exists. This section uses the algorithm given by Gilbert Baumslag and Tekla Taylor[7] to find the center of a one-relator group to analyze the results.

Let $G = \langle X, R \rangle$ be a one-relator group. If $|X| > 2$, then the center is trivial and if G is given on a single variable, then it is cyclic, so its center is G . Therefore, we assume that $|X| = 2$. We also assume, without loss of generality, that one of the variables occur with exponent sum zero.

Let H be the normal closure of b in G and put $b_k = a^{-k}ba^k$. $H = \langle \dots, b_{-1}, b_0, b_1, \dots \rangle$. In the case of a specific relator, $H = \langle b_\lambda, \dots, b_\mu \rangle$ where λ and μ are the minimum and maximum subscripts.

The algorithm has the following steps:

- 1) Use the transformation $b_k = a^{-k}ba^k$ to rewrite $R = R(a, b)$ in terms of the generators b_k .
- 2) Let R_0 be R rewritten in terms of the b_k . Let λ and μ be the maximum and minimum subscripts of the b 's or b^{-1} 's. Count the number

of b 's (with exponent 1 or -1) with minimal or maximum subscript. If they both occurred once then do step A, otherwise step B, where steps A and B are defined as follows:

Step A. If $\lambda + 1 = \mu$, then $H = H'$ since $H = \langle b_\lambda, b_\mu \rangle$, the relator is $b_\alpha b_\beta = 1$ and so H is abelian. In this case, we assume that the center, if not trivial is generated by $a^r h$ where $h \in H$. To determine whether or not the center is trivial we must check the automorphisms $a^r h a^{-r}$ for $r = 1$ to 3^4 until we find an identity automorphism on H . If none are found the center is trivial. If $\lambda + 1 < \mu$, then same method as above except we use $h \in H/H'$ and we check $a^r h a^{-r}$ from 1 to 3^{n^2} where n is the number of elements in H .

Step B. Test the exponent sum of b in R . In order for G to have a non-trivial center then the exponent sum of b must be zero and the normal closure of a in G finitely generated.

Step A of this problem where H is finitely generated was improved from the Baumslag-Taylor algorithm by Frank Levin[8]. Since b_λ and b_μ occur once each it is possible to solve for $b_\mu = w(b_\lambda, \dots, b_{\mu-1})$. In addition a will induce an automorphism Ψ of H defined by $b_i \Psi = b_{i+1}$ ($\lambda < i < \mu - 2$), $b_{\mu-1} \Psi = w$, and for some z , Ψ^z will be inner. In particular, the automorphism Ψ_{ab} which Ψ induces on H abelianized will have order r , as will the $n \times n$ -matrix M which represents Ψ_{ab} . The characteristic

polynomial $F(\gamma)$ of M can be found as follows. First write w additively. Next, replace each b_i in w by γ^i . $F(\gamma)$ is then given by $F(\gamma) = \gamma^n - w$. Since $M^r = I$ only if all zeroes of $F(\gamma)$ are roots of unity, there is an algorithm for finding whether or not r is finite. G will have a non-trivial center if M has finite order r .

The improvement given in this paper is in step B, for which I prove that if one is at this stage and the number of b 's is greater than or equal to 2 then G has a trivial center.

Chapter 2

Decomposability of One-Relator Groups

Positive One-Relator Groups

In this section, we will concentrate on positive one-relator groups (defined below) since they are easier to deal with and give insights into the decomposability question of all one-relator groups.

Definition 19 *A one-relator group is positive if its relator has no negative exponents.*

Most decomposable strings have either one or two suitable transformations. Strings of length divisible by 3 have larger variances due to the strings of form $(ABC)^m$, each of which have 6 transformations. This fact is demonstrated by the following table which is the result of a program which emulates Shenitzer's algorithm.

Table 1

STATISTICS FOR SHENITZER'S ALGORITHM

Positive one-relator groups

$l(R)$	DECOMP. GROUPS	TRANS.	TRANS./STRING	VARIANCE
3	1	6	6	0
4	3	7	2.33	.22
5	2	4	2	0
6	7	17	2.42	2.53
7	4	8	2	0
8	13	26	2	46
9	9	22	2.4	1.58
10	23	43	1.87	.37
11	18	36	2	0

Lemma 20 *If $G = \langle A, B : (AB)^n \rangle$, $n \in \mathbb{Z}^+$ then G is decomposable.*

Proof. - AB is not minimal by the transformation $A \rightarrow A, B \rightarrow A^{-1}B$ and since the concatenation of non-minimal words is non-minimal we are done. ■

Lemma 21 *If $G = \langle A, B, C : (ABC)^n \rangle$, $n \in \mathbb{Z}^+$ then G is decomposable..(Since ABC is non-minimal).*

Proof- The mapping $A \rightarrow A, B \rightarrow A^{-1}B, C \rightarrow C$ will remove the A 's from the string. Since B always occurs with exponent 1 there will be no A^{-1} 's that do not cancel out an A . ■

Lemma 22 *If $l(R)$ is odd and $l(R) \bmod 3 \neq 0$ then the variance of the number of transformations per decomposable string is zero.*

Proof. - All decomposable groups whose relators are of the type above consist of the concatenation of the following forms:

$$A^n, (BC)^n, (ABC)^n$$

or are isomorphic to such a concatenation. A^n is obviously non-decomposable. $(BC)^n$ and $(ABC)^n$ are non-minimal because of the following transformations.

1*) $A \rightarrow A, B \rightarrow B, C \rightarrow B^{-1}C$ removes the generator B

2*) $A \rightarrow A, B \rightarrow BC^{-1}, C \rightarrow C$ removes the generator C

(In the case of $A^n(BC)^k$ the above transformation is only one needed for any length).

There are non-minimal strings of the indicated length that use different transformations. (However, these strings are isomorphic to other strings that need only the above mappings. For example, take the string of length 7, ABACBAB, with can be shown to be non-minimal by the mappings

$$1) A \rightarrow A, B \rightarrow A^{-1}B, C \rightarrow A^{-1}CA$$

$$2) A \rightarrow AB^{-1}, B \rightarrow B, C \rightarrow BCB^{-1}$$

By conjugating, the string ABCABAB is obtained and by mappings 1* and 2* above, non-minimality is shown. In some cases, it is necessary to conjugate and change the letters.)

The necessity of the length of the string, $l(R)$, for which $l(R) \bmod 3 \neq 0$ is because of the case $(ABC)^m, m > 0$. In this case, the following mappings all demonstrate the non-minimality of the string:

$$A \rightarrow A, B \rightarrow A^{-1}B, C \rightarrow C$$

$$A \rightarrow A, B \rightarrow A^{-1}BA, C \rightarrow A^{-1}C$$

$$A \rightarrow A^{-1}BA, B \rightarrow A^{-1}B, C \rightarrow A^{-1}C$$

$$A \rightarrow A, B \rightarrow B, C \rightarrow B^{-1}C$$

$$A \rightarrow A, B \rightarrow BC^{-1}, C \rightarrow C$$

$$A \rightarrow AC^{-1}, B \rightarrow CBC^{-1}, C \rightarrow C$$

If this particular case is discounted, then the lemma holds for all odd lengths.

This shows that if a one-relator group is of odd length and not of the form $(ABC)^n$, then one only needs to check one of the following mappings:

$$A \rightarrow A, B \rightarrow A^{-1}B, C \rightarrow C$$

$$A \rightarrow A, B \rightarrow B, C \rightarrow B^{-1}C$$

The relator has to be put into the form where it begins with the sequence $A^i B^j C^k$, $i, j > 0$. ■

The following table gives the results using the above improvements on Shenitzer's algorithm. Notice that the number of decomposable strings found is the same as in the previous table and the transformations per string are much less, greatly improving the speed of the calculations.

Table 2

$l(R)$	DECOMP. STRINGS	TRANS./STR	VARIANCE
3	1	1	0
4	3	2.3	3.55
5	2	1	0
6	7	1.7	1.9
7	4	1	0
8	13	1.3	1.1
9	9	1	0
10	23	1.17	0.67
11	18	1	0
12	151	1.04	.12
13	72	1	0
14	26675	1.001874	.002
15	291	1	0

This table also demonstrates that the limit of the variances as $l(R)$ approaches infinity is zero.

Case where the number of generators is > 3 and $l(R)$ is
odd

In this case, there are $N * 7^{N-1}$ possible transformations to check according to Shenitzer's algorithm. This number can be drastically shortened if the string is arranged so that it starts with $X_1^p X_2 \dots$ where $p > 0$ and X_1 is the stable element. The variance of the number of the transformations per decomposable string is still zero but there are $l(R) - 1$ for each. This is because each such mapping removes a different letter with the exception of the generator that appears with exponent greater than one. All positive one-relator decomposable groups have only one generator which occurs with an exponent more than one.

Lemma 23 *All positive decomposable one-relator groups whose relator has odd length can be shown to be decomposable by testing the single transformation,*

$$(**) X_1 \rightarrow X_1(stable), X_2 \rightarrow X_1^{-1} X_2, X_3 \rightarrow X_3, \dots, X_m \rightarrow X_m$$

removing the stable letter. If X_1 occurs with an exponent greater than 1 then apply the mapping

$$X_1 \rightarrow X_k, X_k \rightarrow X_1$$

where X_k is a generator that occurs with exponent 1

.

Results with strings of even length with 3 generators

Strings of the form $(AB)^m$, $m > 1$, all have 2 transformations one of which is

$$A \rightarrow A, B \rightarrow A^{-1}B, C \rightarrow C$$

Strings of the form $(ABC)^m$, m even, also have the above, as one of its transformations.

Definition 24 *A sample of a set is finite selection such that any element can be repeated a finite number of repetitions.*

Lemma 25 *All decomposable positive one-relator groups have one or more of the following transformations:*

- 1) $A \rightarrow A, B \rightarrow A^{-1}B, C \rightarrow C$
- 2) $A \rightarrow A, B \rightarrow A^{-1}B, C \rightarrow CA$
- 3) $A \rightarrow A, B \rightarrow A^{-1}B, C \rightarrow CA^{-1}$

Proof. -One of the requirements that a positive string is decomposable is that there is at least one generator that has 1 as its greatest exponent.

Strings of even length that are decomposable have the following form:

$$(X_1X_2)^m(\text{sample of } \{X_3, (X_3X_2), (X_3X_1)\})$$

All substrings of the above are shown to be decomposable by the transformations 1, 2, 3, all of which remove the variable X_1 . ■

Case where the number of generators is > 3

By generalizing the above results, it can be shown that all positive decomposable one-relator groups are formed by concatenating forms of the following types:

$$(X_1 X_2)^j (X_k (\text{sample of the set}(X_2, \dots, X_n))^m \quad 2 \leq k \leq n, j > 0, m \geq 0$$

Since in all mappings the stable letter will be X_1 , the mappings of the other variables, if not identities, will involve X_1 or X_1^{-1} . The mappings will be of the forms:

$$X_1 \rightarrow X_1$$

$$X_2 \rightarrow X_1^{-1} X_2$$

$$X_k \rightarrow X_k X_1 \text{ (if } X_k \text{ occurs to the left of } X_2)$$

$$X_j \rightarrow X_j X_1^{-1} \text{ (if } X_j \text{ occurs to the left of } X_1)$$

$$X_g \rightarrow X_g \text{ (for all other variables).}$$

Therefore there are $3(n-2)$ (number of different $X_1 X_h$ pairs) transformations to check for decomposability of positive one-relator groups (Assuming a correct choice of stable letter)

Example 26 Take the group $G = \langle a, b, c, d : abcbad \rangle$. We will choose a as the stable letter. It is a reasonable candidate, since it doesn't appear with an exponent greater than one. There are in the relator

two different pairs with the stable letter on the left, ab and ad . So we look upon this relator as the concatenation of the strings $abcb$ and ad . In the former we use the above rules to obtain the following mappings:

$$a \rightarrow a$$

$$b \rightarrow a^{-1}b$$

$$c \rightarrow ca$$

For the latter we obtain:

$$a \rightarrow a$$

$$d \rightarrow a^{-1}d$$

The union of the two sets is the appropriate transformation to show the decomposability of the group. (If this union were not possible due to inconsistent mappings then that transformation fails to show decomposability). The number of tries a computer would have to do until it is sure the group is non-decomposable is $3(4-2)(2) = 12$ assuming it made the correct choice of stable letter. If we include random selection of the stable letter then the number of tries will be $12(4) = 48$, 4 being n .

Statistics for Decomposable 3-generator One-relator Groups

Table 3

$l(R)$	DECOMP.	TRANS.	TRANS./STR.	VARIANCE
3	2	12	6	0
4	9	21	2.333333	0.222222
5	23	47	2.043478	3.606805
6	113	286	2.530973	4.443731
7	279	1087	3.896057	8.22934
8	1416	6172	4.358757	7.922141
9	4153	20670	4.97	8.001687
10	14672	76226	5.19	7.067895

As can be seen by the above table, the number of transformations per string increase at a very slow rate as does the variance. Due to the enormous amount of computing time necessary it is not now feasible to test this assumption for much higher values of n . My belief is that the above assumptions will hold out. Unfortunately, while inspecting the output of the program Marvin2(see appendix) the transformations seem equidistributed among the possible transformations. So no probabilistic ordering of transformations appears possible.

Turing machine representation of Shenitzer's Algorithm

Definition 27 *A Turing Machine is a four-tuple*

$$M = (Q, T, P, q_i)$$

in which Q is a finite set of controls states including an initial state q_i , T is an alphabet of tape symbols, and P is a finite program of instructions. Each instruction of P has one of the following forms:

$$right(t) \rightarrow (t', q')$$

$$left(t) \rightarrow (t', q')$$

where $t, t' \in T \cup \{\#\}$ ($\#$ is the blank) and $q, q' \in Q$. Both of these types simply mean that the pointer on the tape can go either left or right one space and at that time the machine will change to a different state.

Definition 28 *A configuration of a Turing machine is a four-tuple*

$$(\varphi, q, t, \psi), \varphi\psi \in (T \cup \{\#\})^*, t \in T \cup \{\#\}, q \in Q$$

in which the rightmost symbol of ψ is not $\#$. The string of symbols $\varphi t \psi$ is called the tape of the configuration. If $\varphi = \lambda$ and $q = q_i$, the configuration is an *initial configuration* of the Turing machine.

Definition 29 *A Deterministic Turing Machine is one where each possible configuration can have at most one production.*

The above description is for a single tape Turing machine. A multi-tape Turing machine is one in which the configuration describes the state of all tapes involved.

Shenitzer's algorithm can be demonstrated by using a four-tape Turing machine. Two will be used as stacks and one will be used as a counter and one will retain the original string. We will designate two of the stacks prefix and suffix. This machine will have $N * 7^{N-1}$ states corresponding to each of the possible transformations. The counter will count the number of variables in the string after mapping is done. .

Starting configuration

input tape = $\wedge$ (input string) {pointer at end of string}

prefix tape = \$(input string) {pointer at end of string}

suffix tape = &(empty)

counter tape = #(empty)

Assume that the input string meets the criteria established in Shenitzer's algorithm. The initial configuration is the input string on the input and prefix stacks and the prefix and counter stacks empty (except for the initial symbols). The finite control will consist of the following three sets of commands:

First set- Checks the number of variables in the input string.

Second set- Apply all $N * 7^{N-1}$ transformations.

Third set - Determines if the number of variables in a string after transformations is less than the original string.

Since none of the above operations involve all of the tapes described above, the Turing machine will be divided into three separate ones corresponding to the sets above.

The initial Turing machine will count the number of variables in the input string. This will involve the input and counter tapes. The configuration of this machine is

$$(\Phi, q_i, \Sigma, \Psi)$$

Where

Φ is the string to the left of the pointer

q_i is an element of Q the the set of states for the first Turing machine.

Σ represents the variable that the pointer is at

Ψ is what is to the left of the pointer on the counter tape.

In order to make the control finite it is necessary to limit the number of variables to a finite number. If the initial string has less variables then the Turing machine will still work. The number of states needed will be $\sum_{r=0}^n P(n, r)$, where n is the number of variables. This procedure could also be done with less states but with more steps by scanning the

input string for a specific variable each time. This will take at most $n * len(\text{input})$ steps and will need $n + 1$ states.

The initial configuration is $(^w, q_i, \Sigma, \Psi)$. The states change according to the order that the variables are encountered in the input string from right to left. For example, the input string, *abccabca*, will use the states $q_i, q_a, q_{ac}, q_{acb}$. Every time a new variable is encountered the pointer on the counter tape is moved one space to the right.

The Turing machine that demonstrates the second set will only use the prefix and suffix tapes. The states will be those pertaining to the currently checked transformation. Essentially this machine pops the stack prefix, applies the transformation to that variable and pushes this onto the stack suffix.

The productions for this machine are of the form

$$(\Phi, q_{x_i}, x_i, \Psi) \rightarrow (\Phi, q, \text{blank}, \Psi f(x_i))$$

where Φ is the string to the left of the pointer on the prefix string. Ψ is the string on the suffix tape. X_i is the variable that the pointer is at. This will halt when the prefix tape only has \$.

Also needed is a machine to eliminate substrings of the kind $X_i X_i^{-1}$ from the suffix tape. The set of states will be designated by the variable being scanned. The productions will be of the form

$$(\Phi X_i X_i^{-1}, q_{x_i}, X_i^{-1}, \Psi) \rightarrow (\Phi, q_{\Sigma(pos-2)}, \Sigma(pos-2), \Psi)$$

where $\Sigma(pos-2)$ represents the variable 2 positions behind X_i . There are $2N$ states in this Turing machine. One for each variable and its inverse. At this stage, there is a reduced word in the suffix tape.

The last Turing machine needed is one that will check for the number of variables on the suffix tape. This will use that tape and the counter tape. The number of states necessary will be 2^n , n being the number of variables in the input string.

An Application to the Thue-Morse Sequence

The Thue-Morse[9] sequence is defined by the morphism

$$\mu : A^* \rightarrow A^*$$

$$\mu(a) = ab, \mu(b) = ba$$

where A denotes the fixed two-letter alphabet $A = \{a, b\}$.

We can define a Thue-Morse sequence as starting with ab and the subsequent elements generated by successive applications of μ . The sequence begins:

ab

abba

abbabaab

abbabaabbaababba

abbabaabbaababbabaababbaabbabaab

It can be seen that except for the initial string ab , all one-relator groups with these strings as relators are not decomposable. This is because none of the strings have a generator where the largest occurring exponent is one.

Chapter 3

An improved algorithm for determining if a one-relator group has a trivial center

First, we will prove the following refinement on the part of the Baumslag-Taylor algorithm that determines if a one-relator group has a trivial center.

Theorem 30 *Let $G = \langle a, b : R(a.b) = 1 \rangle$. Let H be the normal closure of b in G . If H is not finitely generated and the number of b 's is greater than 2 then G has trivial center.*

Proof. By a theorem of Baumslag and Taylor, G will have a trivial center if either the exponent sum of b in R is not zero, or if the exponent sum of b in R is zero and the normal closure of a is not finitely generated.

Assume that the exponent sum of b in R is zero and the number of b 's is greater than 2. We only consider the cases where there are more

than 2 b 's to avoid the exceptions

$$r = b^\varepsilon a^{\zeta(n-2)/2} b^{-\varepsilon} a^{-\zeta(n-2)/2} \text{ where } \varepsilon, \zeta = \pm 1$$

The normal closure of a will be finitely generated as shown by the transposition R after reversing the a 's and b 's. The form $a_0 a_{-\zeta(n-2)/2}$ results, which do not have trivial center because the normal closure of a is finitely generated. We only need examine relators with even length. If the number of b 's is $n-2$ where n is the length of the string, the relator will be of the form

$$r = b^{\varepsilon(n-2)/2} a^\zeta b^{-\varepsilon(n-2)/2} a^{-\zeta} \text{ where } \zeta, \varepsilon = \pm 1.$$

The normal closure of a will not be finitely generated. The transposition that can result after reversing the a 's and b 's is $a_{(n-2)/2}^\zeta a_0^\zeta$. If the number of b 's is 4 (which of course means that there are $n-4$ a 's), after the transpositions, the result will have $4b_\zeta$'s. Since H is not finitely generated, one of λ and μ is not equal to one. In order for the normal closure of a to be finitely generated there are four possible cases:

$$b_\lambda^{\xi_1} b_\alpha^{\xi_2} b_0^{\xi_3} b_\mu^{\xi_4} \text{ or } b_\lambda^{\xi_1} b_0^{\xi_2} b_\alpha^{\xi_3} b_\mu^{\xi_4} \text{ or } b_0^{\xi_1} b_\alpha^{\xi_2} b_\beta^{\xi_3} b_\mu^{\xi_4} \text{ or } b_\lambda^{\xi_1} b_\alpha^{\xi_2} b_\beta^{\xi_3} b_0^{\xi_4}$$

where the $\xi_i = \pm 1$, $\lambda < \alpha < \beta < \mu$ and 2 of the ξ_i 's are negative.

b_0 must appear in every transformation to avoid cancellation of a 's with a^{-1} 's or when it is the largest or smallest subscript. Since the

original string has $(n-4)/2$ $a's$ and $(n-4)/2$ $a^{-1's}$, $|\lambda| + |\alpha| + |\beta| + |\mu| = (n-4)/2$, so after the transformations $b^{-k}ab^k \rightarrow a_k$ are used on the string the result will have $(n-4)/2$ $a'_k's$ and $(n-4)/2$ $a_k^{-1's}$. (The sum of the absolute values of the subscripts is equal to 2.) ■

In looking at the following table, it becomes evident that the percentage with H finitely generated approaches 0 as $l(R) \rightarrow \infty$,and this group contains all of the groups with a non-trivial center. This verifies the intuitive notion that as the length of the relator increases the probability of a one-relator group having a non-trivial center decreases.

Table 4

Statistics for the trivial center of
one-relator groups with H not
finitely generated

$l(R)$	H f.g.	H not f.g	% H not f.g.	% with H not f.g with trivial center $\#b's > 2$
--	----	-----	-----	
4	4	0	0	100
5	0	8	100	100
6	4	12	75	100
7	16	24	60	100
8	36	74	67.3	100
9	80	192	71.6	100
10	132	560	81.9	100
11	225	1585	87.6	100
12	489	4351	89.9	100
13	1159	11735	91	100
14	2906	31783	91.5	100

Chapter 4

Pattern matching with an Ordered Set of Keywords

In the previous chapter, the algorithm used to determine if a one-relator group had a trivial center needed to find occurrences of substrings of the form $a^{-n}b^{\varepsilon k}a^n$ where $n, k \in \mathbb{Z}$ and $\varepsilon = \pm 1$. These substrings had to be found in a particular order, that of strings of longer length before shorter. This brings up an extension of the problem of finding a match from a set of keywords.

Two of the most important algorithms to match a keyword from a set, Aho-Corasick[10] and Commentz-Walter[11]. They are similar in ideas except that Commentz-Walter looks at the words in reverse order. First I will give a description of the Aho-Corasick algorithm, then I will present a modification, when the keywords have a prescribed order. The straightforward way to look for multiple keywords in an input string is to apply the fastest single keyword pattern-matching once for each keyword. (All of the main ones run in $O(m + n)$ time, where m is

the length of the pattern and n is the length of the input string.) This would give a solution whose running time is $O(m + kn)$, where k is the number of keywords and m is the sum of the lengths of the keywords. The Aho-Corasick algorithm solves this problem in $O(m + n)$ time. The general idea is that it first constructs from the set of keywords a pattern-matching automaton in $O(m)$ time. The automaton then reads the input string and looks for all the keywords simultaneously in $O(n)$ time. The automaton is similar to a deterministic finite automaton except that it has two transition functions, a forward and failure function. The failure function is used only when the forward function fails.

The Aho-Corasick algorithm has the following components:

Q : a finite set of states

Σ : a finite input alphabet

$g : Q \times \Sigma \rightarrow Q \cup \{\text{fail}\}$, a forward transition function

$h : Q \rightarrow Q$, a failure transition function

F : a set of accepting states

The pseudocode for this algorithm is:

begin

$q := q_0$

for $j := 1$ *to* n *do*

begin

while $g[q, s_j] = fail$ *do* $q := h[g]$

$q := g[q, s_j]$

if q *is in* F *then return* "yes"

end

return "no"

end.

It is easier to understand with an example. Let $\{abc, bbc, aab, ba\}$.

This set of keywords result in the following trie:

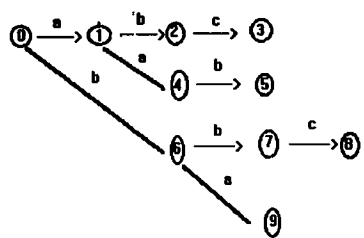


Figure 4.1:

This can also be represented by the following table

state	a	b	c	failure	transiton
0	1	6	fail		0
1	4	2	fail		0
2	fail	fail	3		6
3	fail	fail	fail		0
4	fail	5	fail		1
5	fail	fail	fail		2
6	9	7	fail		6
7	fail	fail	8		0
8	fail	fail	fail		0
9	fail	fail	fail		1

In the problem with an ordered set of keywords, the situation is more complicated since after we find a keyword we must remove it from the string lest it be a substring or contain a prefix or suffix that is a substring of a keyword, so we can continue searching. The psuedocode for this is:

begin

$L := 1;$

begin

$q := q_0;$

```

for  $j := 1$  to  $n$  do
begin
while  $g[q, s_j] := fail$  do  $q := h(q)$ ;

 $q := g[q, s_j]$ ;

if  $q$  is in  $F$  then
begin
return "yes" for  $W_L$ ;

for  $r := j$  to  $n$  do

 $s_{r-length(W_L)+1} = s_{r+1}$ ;

for  $p := n$  down to  $n - length(W_L)$  do

 $s_p := blank$ ;

end;

end;

 $L := L + 1$ ;

end.

```

This algorithm is $O(M+LN)$ where L is the number of keywords to check , N is the size of the input string and M is the sum of the lengths of the keywords. The automaton necessary to recognize keywords takes $O(M)$ to formulate and the above program runs in $O(LN)$. The brute force algorithm for pattern-matching of a set of keywords without order is also $O(M+LN)$.

Appendices

The following is the pseudocode for the computer program to emulate

Shenitzer's algorithm:

(MAINPROGRAM)

BEGIN

READ(LEN, INV);

IF INV="Y" THEN

BEGIN

INVSU = 'Y';

HIGHSYMBOL = 6;

END

ELSE

HIGHSYMBOL = 3;

I = 1;

WHILE I < LEN DO

BEGIN

```

S[I] = 'A';
I = I + 1;
END
ALLDONE = 'NO';
WHILE ALLDONE = 'NO' DO
BEGIN
ENDLOOP = 'NO';
WHILE ENDLOOP = 'NO';
INCREMENT;
IF (ALLDONE = 'NO') THEN
IF(INV = 'Y ')
THEN CHECKPATTERN
ELSE
NICHECKPATTERN;
ELSE ENDLOOP = 'YES'
IF STATUS = 'MAYBE'
THEN ENDLOOP = 'YES';
SPRINT(STR);
TRANSLATE;
WRITE('NUMBER OF DECOMPOSABLE STRING =');
WRITELN(DECOMCOUNT);

```

WRITELN('TOTAL TRANSFORM, RESULT. IN DECOMPOSITION =');

WRITELN(TRANSCOUNT);

**WRITELN('TRANSFORMATION/STRING=');WRITELN
(SUMOFSQUARES - (TRANSCOUNT)*2/DECOMCOUNT);**

.WRITELN(TRANSCOUNT/DECOMCOUNT);

WRITELN('ESTIMATED VARIANCE OF ABOVE =');

In the above pseudocode the following procedures are used:

INCREMENT- This changes the string that is being checked for decomposability.

CHECKPATTERNS - This checks patterns in the candidate string that will disqualify it for consideration and therefore not be checked.

These conditions are:

a) Not beginning with the sequence $A^n B$ or $A^n B^{-1}$

b) Ending with an A or an A^{-1}

c) The absolute power of all occurrences of A, B or C are greater than one.

d) The string is not cyclically reduced.

NICHECKPATTERNS - Same as above , but only used when no inverses are in the original string, it is not necessary for the program but it speeds up computing time.

SPRINT-Translates string back from its numerical code and prints it.

TRANSLATE - Check all the 147 possible transformations and prints out those strings plus the transformations that result in a decomposable group.

```
PROCEDURE INCREMENT;
VAR
POSITION,DONE: INTEGER;
BEGIN
IF DECOMFLAG = 'YES' THEN
BEGIN
STORE(STR);
DECOM = DECOM + 1;
SUMOFSQUARES = SUMOFSQUARES + TRANSFORM*2;
TRANSFORM = 0;
END;
DECOMFLAG = 'NO';
WHILE DONE = 'NO';
BEGIN
IF (S[POSIT]) < HIGHSYMBOL) THEN
BEGIN
```

```

S[POSIT] = S[POSIT] + 1;
DONE = 'YES';
END
ELSE
IF POSITION = S[0] THEN
BEGIN
ALLDONE = 'YES';
DONE = 'YES';
ELSE
BEGIN
S[POSIT] = 1;
POSITION = POSITION + 1;
END
VHOLD = VARIABLE = VARIABLECOUNT(S);
END;

```

This procedure stores the decomposable strings, keeps track of their number, calculates the sum of the squares of the number of the number of transformations that result in decomposition. Its main function however is to change strings to be considered it does this in numerical order starting with a string of A 's to a string of C^{-1} 's. This ordering is due to the designations of $A = 1, A^{-1} = 4, B = 2, B^{-1} = 5, C = 3, C^{-1} = 6$. The

above procedure has the procedures STORE and VARIABLECOUNT.

STORE-stores the decomposable string and its isomorphic images.

VARIABLECOUNT-counts number of variables in the string since if there is one variable the string is obviously not decomposable. If there are two then the mappings of C can be ignored.

```
PROCEDURE CHECKPATTERNS;
VAR
COUNT, I, PREV, STATE, STATUS : INTEGER;
BEGIN
STATUS = AAABCHECK;
IF STATUS = 'MAYBE' THEN
WHILE I<LEN DO
IF ABS(S(I) - S(I +1)) = 3 THEN STATUS = 'NO';
IF STATUS  $\neq$  'NO' THEN
IF S(I)  $\neq$  S(LEN) THEN STATUS ='MAYBE'
ELSE
BEGIN
STATE = 1;
PREV = S(1);
STATUS = 'MAYBE';
WHILE I < LEN DO
```

```

IF STATE = 1 THEN
IF (S(I) = B OR S(I) = B-1 ) THEN
BEGIN
STATE = 2;
PREV = S(I);
COUNT = 1;
END
ELSE IF STAE = 2 THEN
BEGIN
IF S(I) = PREV OR ABS(S(I) - PREV) = 3 THEN
COUNT = COUNT + 1;
ELSE
IF COUNT = 1 THEN STATUS = 'MAYBE';
COUNT = COUNT + 1;
ELSE
IF COUNT = 1 THEN STATUS = 'MAYBE';
COUNT = 1;
PREV = S(I);
END
IF STATUS = 'MAYBE';
THEN

```

```

BEGIN

COUNT = 0;

PREV = S(I);

DECOMPOSABLE = 'NO';

WHILE I < LEN

BEGIN

IF (S(I) = PREV OR ABS(PREV-S(I)) = 3) THEN

COUNT = COUNT + 1;

ELSE

IF COUNT = 1 THEN DECOMPOSABLE = 'MAYBE';

I = I+1;

END;

IF COUNT = 1 THEN DECOMPOSABLE = 'MAYBE';

I = I + 1;

END

IF (STATUS ≠ 'NO') THE STATUS = CHECKSTRING(S);

END

```

The above procedure involves the procedures CHECKSTRING and AAABCHECK.

CHECKSTRING compares the string with the previously stored decomposable strings and their isomorphic images.

AAABCHECK makes sure the string starts with the sequence $AAA....B$

or $AAA....B^{-1}$.

PROCEDURE SPRINT;

VAR

I: INTEGER;

BEGIN

WHILE I $\leq$ LEN DO

BEGIN

IF X(I) = 1 THEN PRINT 'A';

IF X(I) = 2 THEN PRINT 'B';

IF X(I) = 3 THEN PRINT 'C';

IF X(I) = 4 THEN PRINT 'A*';

IF X(I) = 5 THEN PRINT 'B*';

IF X(I) = 6 THEN PRINT 'C*';

I = I + 1;

END;

END;

PROCEDURE TRANSLATE;

BEGIN

STABLE = 'A'; STABLEI = 'A*';

SEC = 'B'; SECI = 'B*';

```

THR = 'C' ; THRI ='C*';
SETUPTRANS;
STABLE = 'B'; STABLEI ='B*';
SEC = 'A'; SECI = 'A*';
THR = 'C' ; THRI = 'C*';
SETUPTRANS;
STABLE = 'C'; STABLEI = 'C*';
SEC = 'A'; SECI = 'A*';
THR = 'B' ; THRI = B*';
SETUPTRANS;
END;

```

The procedure `setuptrans` set up all the transformations with the setting of the stable letter. It sets up all the 147 possible transformations.

The following C++ program uses Shenitzer's algorithm to find decomposable one-relator groups with the input being the length of the relator..

```

/* MARVIN2 */

#include <math.h>
#include <constrea.h>
#define NO 0
#define MAYBE 1
#define YES 2
#define A 1
#define A_ 4
#define B 2
#define B_ 5
#define C 3
#define C_ 6

int alldone=NO,decomposable,HIGHSYMBOL,n2,s1[32],s2[32],s3[32];
int s,si,vhold,v1,vlo[10],vli,vlio[10],v2,v2o[10],v2i,v2io[10];
char inverse_switch='Y';
int decomposable_flag=NO,decomposable_count=0;
float sum_of_squares=0.0,transformation_count=0.0,transforms_per=0.0;
extern int check_string(int a[32]);
extern void store_all(int a[32]);
extern charprint(int x);
extern sprint(int a[32]);

int AAA_B_check()
{
    int stop_loop=NO,okay,p=2;

    if (s1[1]!=A & s1[1]!=A_) okay=NO;
    else
        while (stop_loop==NO)
        {
            if (s1[p] == C | s1[p] == C_) // C encountered first
            {
                okay=NO;
                stop_loop=YES;
            }
            else if (s1[p]==B | s1[p]==B_)
            {
                okay=MAYBE;
                stop_loop=YES;
            }

            p++;
            if (p>s1[0]) // B never encountered
            {
                okay=NO;
                stop_loop=YES;
            }
        }
    return(okay);
}

int ni_AAA_B_check()
{
    int stop_loop=NO,okay,p=2;

    if (s1[1]!=A ) okay=NO;
    else

```

```

while (stop_loop==NO)
{
    if (s1[p] == C ) // C encountered first
    {
        okay=NO;
        stop_loop=YES;
    }
    else if (s1[p]==B )
    {
        okay=MAYBE;
        stop_loop=YES;
    }

    p++;
    if (p>s1[0]) // B never encountered
    {
        okay=NO;
        stop_loop=YES;
    }
}
return(okay);
}

```

```

int check_patterns()
{ //1
    int count,i,prev,state,status;

    status=AAA_B_check();
    if (status==MAYBE)
    { //2
        for (i=1;i<s1[0];i++) if (abs(s1[i] - s1[i+1])==3) status=NO;
        if (s1[s1[0]]==A | s1[s1[0]]==A_) status=NO;
        // if last position is A or A* the string is discarded

        if (status!=NO)
        {
            if (s1[1]!=s1[s1[0]]) status=MAYBE;
            else
            { //3
                state=1;prev=s1[1];status=NO;
                for (i=1;i<s1[0];i++)
                { //4
                    if (state==1)
                    { //5
                        if (s1[i]==B | s1[i]==B_)
                        { //6
                            state=2;
                            prev=s1[i];
                            count=1;
                        } //6
                    } //5
                    else if (state==2)
                    { //6
                        if (s1[i]==prev | abs(s1[i]-prev)==3) count++;
                        else
                        { //7
                            if (count==1) status=MAYBE;
                        }
                    }
                }
            }
        }
    }
}

```

```

        count=1;prev=s1[i];
    } //7
} //6
if (status==MAYBE)
{ //7
    count=0; prev=s1[1]; decomposable=NO;
    for (i=1;i<=s1[0];i++)
    { //8
        if (s1[i]==prev | abs(prev-s1[i])==3) count++;
        else if (count==1) decomposable = MAYBE;
    } //8
    if (count==1) decomposable = MAYBE;
} //7
} //6
} //5
} //4
}
if (status!=NO) status=check_string(s1);
return(status);
} //1

int ni_check_patterns()
// SEPERATE ROUTINE IF NOT CHECKING INVERSES
{ //1
    int count,i,prev,state,status;

    status=ni_AAA_B_check();
    if (status==MAYBE)
    { //2
        //for (i=1;i<s1[0];i++) if (abs(s1[i] - s1[i+1])==3) status=NO;
        if (s1[s1[0]]==A | s1[s1[0]]==A_) status=NO;//if (abs(s1[1]-s1[s1[0]])==3) s
        if (status!=NO)
        {
            if (s1[1]!=s1[s1[0]]) status=MAYBE;
            else
            { //3
                state=1;prev=s1[1];status=NO;
                for (i=1;i<s1[0];i++)
                { //4
                    if (state==1)
                    { //5
                        if (s1[i]==B )
                        { //6
                            state=2;
                            prev=s1[i];
                            count=1;
                        } //6
                    } //5
                    else if (state==2)
                    { //6
                        if (s1[i]==prev ) count++;
                        else
                        { //7
                            if (count==1) status=MAYBE;
                            count=1;prev=s1[i];
                        } //7
                    } //6
                } //6
                if (status==MAYBE)
                { //7
                    count=0; prev=s1[1]; decomposable=NO;

```

```

        for (i=1;i<=s1[0];i++)
        { //8
            if (s1[i]==prev ) count++;
            else if (count==1) decomposable = MAYBE;
        } //8
        if (count==1) decomposable = MAYBE;
    } //7
} //6
} //5
} //4
}
/*if (status==NO)
{
    sprint(s1);
    cout<<" Pattern cannot be decomposable\n";
    cin>>z;
    if (z=='q') alldone=YES;
} */
if (status!=NO) status=check_string(s1);
return(status);
} //1

void print_decomposable()
{
    cout<<"in=";
    sprint(s1);
    cout<<" s=";
    charprint(s);
    cout<<" v1=";
    sprint(v1o);
    cout<<" v2=";
    sprint(v2o);
    cout<<" >> ";
    sprint(s2);
    cout<<" >> ";
    sprint(s3);
    cout<<"\n";
}

int variable_count(int x[100])
{
    int a=0,b=0,c=0,i,k;
    for (i=1;i<=x[0];i++)
    switch(x[i])
    {
        case 1: a=1;break;
        case 2: b=1;break;
        case 3: c=1;break;
        case 4: a=1;break;
        case 5: b=1;break;
        case 6: c=1;break;
    }
    return(a+b+c);
}

// VARIABLE HIGHSYMBOL FOR NOT CHECKING INVERSES

void increment_s1()

```

```

{
    int position=2,done=NO;

    if (decomposable_flag==YES)
    {
        store_all(s1);
        decomposable_count++;
        sum_of_squares+=transforms_per * transforms_per;
        transforms_per=0;
    }
    decomposable_flag=NO;
    while(done==NO)
    {
        if (s1[position]<HIGHSYMBOL)
        {
            s1[position]++;
            done=YES;
        }
        else
        {
            if (position==s1[0])
            {
                alldone=YES;
                done=YES;
            }
            else
            {
                s1[position]=1;
                position++;
            }
        }
    }
    vhold=variable_count(s1);
    /* sprint(s1);
    cout<<"\n"; */
}

void int_copy(int x)
{
    s2[0]++;
    s3[0]++;
    s2[s2[0]]=x;
    s3[s3[0]]=x;
    /* to eliminate consecutive inverses */
    if (s3[0]>1 & abs(s3[s3[0]]-s3[s3[0]-1])==3)s3[0]--2;
}

void string_copy(int x[10])
{
    int j,k;
    for (j=1;j<=x[0];j++)
    {
        s2[0]++;
        s3[0]++;
        s2[s2[0]]=x[j];
        s3[s3[0]]=x[j];
        /* to eliminate consecutive inverses */
    }
}

```

```

        if (s3[0]>1 & abs(s3[s3[0]]-s3[s3[0]-1])==3)s3[0]--;
    }
}

void transform_s1()
{
    int i,v,x;

    s2[0]=0;
    s3[0]=0;

    for (i=1;i<=s1[0];i++)
    {
        x=s1[i];
        if (x==si) int_copy(si);
        else if (x==v1i) string_copy(v1io);
        else if (x==v2i) string_copy(v2io);
        else if (x==s) int_copy(s);
        else if (x==v1) string_copy(v1o);
        else if (x==v2) string_copy(v2o);
    }
    v=variable_count(s3);
    if (v<vhold)
    {
        print_decomposable();
        decomposable_flag=YES;
        transformation_count++;
        transforms_per++;
    }
    /*else
    {
        sprintf(s1);
        cout<<">>";
        sprintf(s3);
        cout<<" Not decomposable\n";
    } */
}

void set_second_variable()
{
    /* C=>C */    v2o[0]=1;v2o[1]=v2;    v2io[0]=1;v2io[1]=v2i;transform_s1();
    if (vhold>2)
    {
        /* C=>AC */    v2o[0]=2;v2o[1]=s;v2o[2]=v2;    v2io[0]=2;v2io[1]=v2i;v2io[2]=si
        /* C=>A*C */    v2o[0]=2;v2o[1]=si;v2o[2]=v2;    v2io[0]=2;v2io[1]=v2i;v2io[2]=s;
        /* C=>CA */    v2o[0]=2;v2o[1]=v2;v2o[2]=s;    v2io[0]=2;v2io[1]=si;v2io[2]=v2i
        /* C=>CA* */    v2o[0]=2;v2o[1]=v2;v2o[2]=si;    v2io[0]=2;v2io[1]=s;v2io[2]=v2i;
        /* C=A*CA */    v2o[0]=3;v2o[1]=si;v2o[2]=v2;v2o[3]=s;
                        v2io[0]=3;v2io[1]=s;v2io[2]=v2i;v2io[3]=si;transform_s1(
        /* C=>ACA* */    v2o[0]=3;v2o[1]=s;v2o[2]=v2;v2o[3]=si;
                        v2io[0]=3;v2io[1]=si;v2io[2]=v2i;v2io[3]=s;transform_s1(
    }
}

void setup_transformations()
{
    v1o[0]=1;v1o[1]=v1;    v1io[0]=1;v1io[1]=v1i;
    v1o[0]=2;v1o[1]=s;v1o[2]=v1;    v1io[0]=2; v1io[1]=v1i;v1io[2]=si;
    v1o[0]=2;v1o[1]=si;v1o[2]=v1;    v1io[0]=2;v1io[1]=v1i;v1io[2]=s;
}

```

```

        vlio[0]=2;vlio[1]=v1;vlio[2]=s;    vlio[0]=2;vlio[1]=si;vlio[2]=vli;
        vlio[0]=2;vlio[1]=v1;vlio[2]=si;    vlio[0]=2;vlio[1]=s;vlio[2]=vli;
        vlio[0]=3;vlio[1]=si;vlio[2]=v1;vlio[3]=s;
        vlio[0]=3;vlio[1]=s;vlio[2]=v1;vlio[3]=si;    set_second_varia
        vlio[0]=3;vlio[1]=s;vlio[2]=v1;vlio[3]=si;
        vlio[0]=3;vlio[1]=si;vlio[2]=vli;vlio[3]=s;set_second_variable()
    }

void translate_sl()
{
    s=A;si=A_;v1=B;vli=B_;v2=C;v2i=C_;setup_transformations();
    s=B;si=B_;v1=A;vli=A_;v2=C;v2i=C_;setup_transformations();
    s=C;si=C_;v1=A;vli=A_;v2=B;v2i=B_;setup_transformations();
}

void main()
{
    int endloop,i,x;

    clrscr();
    cout<<"C' VERSION\n";
    cout<<"\n";
    cout<<"Press Ctrl+Print Screen now to have results printed.\n";
    cout<<"Please input the number of positions desired:";
    cin>>s1[0];
    cout<<"\nWould you like strings with inverses to be considered (Y/N)?";
    cin>>inverse_switch;
    if (inverse_switch=='y') inverse_switch='Y';
    if (inverse_switch=='Y') HIGHSYMBOL=6; else HIGHSYMBOL=3;

    // QUESTION ABOUT INVERSES

    for(i=1;i<=s1[0];i++) s1[i]= A;
    while (alldone==NO)
    {
        endloop=NO;
        while (endloop==NO)
        {
            increment_sl();
            if (alldone==NO)
            {
                if(inverse_switch=='Y') x=check_patterns();
                else x=ni_check_patterns();
            }
            else endloop = YES;
            if (x==MAYBE) endloop=YES;
        }
        //sprint(s1);cout<<"\n";
        translate_sl();
    }
    cout<<"Number of decomposable strings=";
    cout<<decomposable_count;
    cout<<"\n";
    cout<<"Total transformations resulting in decomposition=";
    cout<<transformation_count;
    cout<<"\n";
    cout<<"Transformations / Strings=";
    cout<<transformation_count/decomposable_count,"\\n";
    cout<<"\n";
    cout<<"Estimated variance of above=";
    cout<< (sum_of_squares - ((transformation_count * transformation_count )

```

```
    ) / decomposable_count;  
    cout<<"\n";  
}
```

Examples of output from the previous program.

3

Would you like strings with inverses to be considered (Y/N)?y

```
in=ABC s=A v1=A*B v2=C >> AA*BC >> BC
in=ABC s=A v1=A*BA v2=A*C >> AA*BAA*C >> BC
in=ABC s=B v1=A v2=B*C >> ABB*C >> AC
in=ABC s=B v1=AB* v2=C >> AB*BC >> AC
in=ABC s=C v1=A v2=BC* >> ABC*C >> AB
in=ABC s=C v1=AC* v2=CB* >> AC*CB*C >> AB
in=AB*C s=A v1=BA v2=C >> AA*B*C >> B*C
in=AB*C s=A v1=ABA* v2=A*C >> AA*B*AA*C >> B*C
in=AB*C s=B v1=A v2=BC >> AB*BC >> AC
in=AB*C s=B v1=AB v2=C >> ABB*C >> AC
in=AB*C s=C v1=A v2=CB >> AB*C*C >> AB*
in=AB*C s=C v1=AC* v2=C*BC >> AC*CB*C*C >> AB*
Number of decomposable strings=2
Total transformations resulting in decomposition=12
Transformations / Strings=6
Estimated variance of above=0
```

Type 'EXIT' to return to program

A:\>marvin2

'C' VERSION

Press Ctrl+Print Screen now to have results printed.

Please input the number of positions desired:3

Would you like strings with inverses to be considered (Y/N)?n

```
in=ABC s=A v1=A*B v2=C >> AA*BC >> BC
in=ABC s=A v1=A*BA v2=A*C >> AA*BAA*C >> BC
in=ABC s=B v1=A v2=B*C >> ABB*C >> AC
in=ABC s=B v1=AB* v2=C >> AB*BC >> AC
in=ABC s=C v1=A v2=BC* >> ABC*C >> AB
in=ABC s=C v1=AC* v2=CB* >> AC*CB*C >> AB
Number of decomposable strings=1
Total transformations resulting in decomposition=6
Transformations / Strings=6
Estimated variance of above=0
```

Type 'EXIT' to return to program

A:\>marvin2

'C' VERSION

Press Ctrl+Print Screen now to have results printed.

Please input the number of positions desired:4

Would you like strings with inverses to be considered (Y/N)?y

```
in=ABAB s=A v1=A*B v2=C >> AA*BAA*B >> BB
in=ABAB s=B v1=AB* v2=C >> AB*BAB*B >> AA
in=ABCB s=A v1=A*B v2=CA >> AA*BCAA*B >> BCB
in=ABCB s=B v1=AB* v2=CB* >> AB*BCB*B >> AC
in=ABCB s=C v1=AC v2=C*B >> ACC*BCC*B >> ABB
in=AB*CB s=B v1=A v2=BCB* >> AB*BCB*B >> AC
in=AB*CB s=B v1=AB v2=CB* >> ABB*CB*B >> AC
in=ABC*B s=A v1=A*B v2=A*C >> AA*BC*AA*B >> BC*B
in=ABC*B s=B v1=AB* v2=BC >> AB*BC*B*B >> AC*
in=ABC*B s=C v1=AC* v2=CB >> AC*CB*CB >> ABB
in=AB*C*B s=B v1=A v2=B*CB >> AB*BC*B*B >> AC*
in=AB*C*B s=B v1=AB v2=BC >> ABB*C*B*B >> AC*
in=AABC s=B v1=A v2=B*C >> AAB*B*C >> AAC
```

```

in=AABC s=C v1=A v2=BC* >> AABC*C >> AAB
in=AB*A*C s=A v1=BA v2=AC >> AA*B*AA*AC >> B*C
in=AB*A*C s=A v1=ABA* v2=C >> AA*B*AA*AC >> B*C
in=AAB*C s=B v1=A v2=BC >> AAB*BC >> AAC
in=AAB*C s=C v1=A v2=CB >> AAB*C*C >> AAB*
in=AB*B*C s=A v1=ABA* v2=A*C >> AA*B*AA*B*AA*AC >> B*B*C
in=AB*B*C s=B v1=AB v2=BC >> ABB*B*BC >> AC
in=AB*B*C s=C v1=AC* v2=C*BC >> AC*CB*C*CB*C*C >> AB*B*
Number of decomposable strings=9
Total transformations resulting in decomposition=21
Transformations / Strings=2.333333
Estimated variance of above=0.222222

```

Type 'EXIT' to return to program
A:\>marvin2
'C' VERSION

Press Ctrl+Print Screen now to have results printed.
Please input the number of positions desired:4

Would you like strings with inverses to be considered (Y/N)?n

```

in=ABAB s=A v1=A*B v2=C >> AA*BAA*B >> BB
in=ABAB s=B v1=AB* v2=C >> AB*BAB*B >> AA
in=ABCB s=A v1=A*B v2=CA >> AA*BCAA*B >> BCB
in=ABCB s=B v1=AB* v2=CB* >> AB*BCB*B >> AC
in=ABCB s=C v1=AC v2=C*B >> ACC*BCC*B >> ABB
in=AABC s=B v1=A v2=B*C >> AAB*B*C >> AAC
in=AABC s=C v1=A v2=BC* >> AABC*C >> AAB
Number of decomposable strings=3
Total transformations resulting in decomposition=7
Transformations / Strings=2.333333
Estimated variance of above=0.222222

```

Type 'EXIT' to return to program
A:\>marvin2
'C' VERSION

Press Ctrl+Print Screen now to have results printed.
Please input the number of positions desired:5

Would you like strings with inverses to be considered (Y/N)?y

```

in=ABCAB s=A v1=A*B v2=C >> AA*BCAA*B >> BCB
in=ABCAB s=B v1=AB* v2=C >> AB*BCAB*B >> ACA
in=ABC*AB s=A v1=A*B v2=C >> AA*BC*AA*B >> BC*B
in=ABC*AB s=B v1=AB* v2=C >> AB*BC*AB*B >> AC*A
in=AB*CCB s=B v1=A v2=BCB* >> AB*BCB*BCB*B >> ACC
in=AB*CCB s=C v1=A v2=C*BC >> ACB*C*CCC*BC >> ACC
in=AB*CCB s=C v1=CA v2=C*BC >> CACB*C*CCC*BC >> CACC
in=AB*CCB s=C v1=C*A v2=C*BC >> C*ACB*C*CCC*BC >> C*ACC
in=AB*CCB s=C v1=AC v2=C*BC >> ACCB*C*CCC*BC >> ACCC
in=AB*CCB s=C v1=AC* v2=C*BC >> AC*CB*C*CCC*BC >> AC
in=AB*CCB s=C v1=C*AC v2=C*BC >> C*ACCB*C*CCC*BC >> C*ACCC
in=AB*CCB s=C v1=CAC* v2=C*BC >> CAC*CB*C*CCC*BC >> CAC
in=AAB*CB s=B v1=A v2=BCB* >> AAB*BCB*B >> AAC
in=AB*B*CB s=B v1=AB v2=BCB* >> ABB*B*BCB*B >> AC
in=AB*A*A*B s=A v1=ABA* v2=C >> AA*B*AA*A*ABA* >> A*
in=AAB*C*B s=B v1=A v2=B*CB >> AAB*BC*B*B >> AAC*
in=AB*B*C*B s=B v1=AB v2=B*CB >> ABB*B*BC*B*B >> AC*
in=AB*C*C*B s=B v1=A v2=B*CB >> AB*BC*B*BC*B*B >> AC*C*
in=AB*C*C*B s=C v1=A v2=CBC* >> AC*B*CC*C*CBC* >> AC*C*

```

```

in=AB*C*C*B s=C v1=CA v2=CBC* >> CAC*B*CC*C*CBC* >> CAC*C*
in=AB*C*C*B s=C v1=C*A v2=CBC* >> C*AC*B*CC*C*CBC* >> C*AC*C*
in=AB*C*C*B s=C v1=AC v2=CBC* >> ACC*B*CC*C*CBC* >> AC*
in=AB*C*C*B s=C v1=AC* v2=CBC* >> AC*C*B*CC*C*CBC* >> AC*C*C*
in=AB*C*C*B s=C v1=C*AC v2=CBC* >> C*ACC*B*CC*C*CBC* >> C*AC*
in=AB*C*C*B s=C v1=CAC* v2=CBC* >> CAC*C*B*CC*C*CBC* >> CAC*C*C*
in=AAABC s=B v1=A v2=B*C >> AAABB*C >> AAAC
in=AAABC s=C v1=A v2=BC* >> AAABC*C >> AAAB
in=AB*A*CC s=A v1=ABA* v2=C >> AA*B*AA*CC >> B*CC
in=AB*B*A*C s=A v1=ABA* v2=C >> AA*B*AA*B*AA*C >> B*B*C
in=AAAB*C s=B v1=A v2=BC >> AAAB*BC >> AAAC
in=AAAB*C s=C v1=A v2=CB >> AAAB*C*C >> AAAB*
in=AB*AB*C s=A v1=BA v2=C >> AA*B*AA*B*C >> B*B*C
in=AB*AB*C s=B v1=AB v2=C >> ABB*ABB*C >> AAC
in=AB*CB*C s=B v1=A v2=BC >> AB*BCB*BC >> ACC
in=AB*CB*C s=C v1=A v2=CB >> AB*C*CB*C*C >> AB*B*
in=AB*B*B*C s=A v1=ABA* v2=A*C >> AA*B*AA*B*AA*B*AA*C >> B*B*B*C
in=AB*B*B*C s=C v1=AC* v2=C*BC >> AC*CB*C*CB*C*CB*C*C >> AB*B*B*
in=ABAAB* s=A v1=ABA* v2=C >> AABA*AAA*B*A >> AAA
in=AB*C*AB* s=A v1=BA v2=C >> AA*B*C*AA*B* >> B*C*B*
in=AB*C*AB* s=B v1=AB v2=C >> ABB*C*ABB* >> AC*A
in=ABBCB* s=B v1=AB* v2=B*CB >> AB*BBB*CB* >> AC
in=AB*AC*B* s=A v1=BA v2=ACA* >> AA*B*AA*C*AA*B* >> B*C*B*
in=AB*AC*B* s=B v1=AB v2=BCB* >> ABB*ABB*C*BB* >> AAC*
in=ABBC*B* s=B v1=AB* v2=BCB* >> AB*BBB*C*BB* >> AC*
in=AB*C*B*C* s=B v1=A v2=CB* >> AB*BC*B*BC* >> AC*C*
in=AB*C*B*C* s=C v1=A v2=C*B >> AB*CC*B*CC* >> AB*B*
in=AB*A*C*C* s=A v1=ABA* v2=C >> AA*B*AA*C*C* >> B*C*C*
Number of decomposable strings=23
Total transformations resulting in decomposition=47
Transformations / Strings=2.043478
Estimated variance of above=3.606805

```

Type 'EXIT' to return to program
A:\>marvin2
'C' VERSION

Press Ctrl+Print Screen now to have results printed.
Please input the number of positions desired:5

Would you like strings with inverses to be considered (Y/N)?n
in=ABCAB s=A v1=A*B v2=C >> AA*BCAA*B >> BCB
in=ABCAB s=B v1=AB* v2=C >> AB*BCAB*B >> ACA
in=AAABC s=B v1=A v2=B*C >> AAABB*C >> AAAC
in=AAABC s=C v1=A v2=BC* >> AAABC*C >> AAAB
Number of decomposable strings=2
Total transformations resulting in decomposition=4
Transformations / Strings=2
Estimated variance of above=0

Type 'EXIT' to return to program
A:\>

The following C++ program uses the faster algorithm to decomposable positive one-relator groups.

```

/* MARVINOE */

#include <math.h>
#include <bcd.h>
#include <constrea.h>
#define NO 0
#define MAYBE 1
#define YES 2
#define A 1
#define A_ 4
#define B 2
#define B_ 5
#define C 3
#define C_ 6
#define HIGHSYMBOL 3

int alldone=NO,decomposable,oddeven,n2,s1[32],s2[32],s3[32];
int s,si,so[10],sio[10],vhold,v1,v1o[10],v1i,v1io[10],v2,v2o[10],v2i,v2io[10];
char inverse_switch='Y';
int decomposable_flag=NO;
bcd decomposable_count=0.0;
bcd sum_of_squares=0.0,transformation_count=0.0,transforms_per=0.0;
extern int check_string(int a[32]),rotate(int[25],int[25]);
extern void store_all(int a[32]);
extern charprint(int x);
extern sprint(int a[32]);

int ni_AAA_B_check()
(
    int stop_loop=NO,okay,p=2;

    if (s1[1]!=A ) okay=NO;
    else
    while (stop_loop==NO)
    (
        if (s1[p] == C ) // C encountered first
        (
            okay=NO;
            stop_loop=YES;
        )
        else if (s1[p]==B )
        (
            okay=MAYBE;
            stop_loop=YES;
        )

        p++;
        if (p>s1[0]) // B never encountered
        (
            okay=NO;
            stop_loop=YES;
        )
    )
    return(okay);
)

```

```

int ni_check_patterns()
// SEPERATE ROUTINE IF NOT CHECKING INVERSES
{ //1
    int count,i,prev,state,status;

    status=ni_AAA_B_check();
    if (status==MAYBE)
    { //2
        //for (i=1;i<s1[0];i++) if (abs(s1[i] - s1[i+1])==3) status=NO;
        if (s1[s1[0]]==A | s1[s1[0]]==A_) status=NO;//if (abs(s1[1]-s1[s1[0]])==3) s
        if (status!=NO)
        {
            if (s1[1]!=s1[s1[0]]) status=MAYBE;
            else
            { //3
                state=1;prev=s1[1];status=NO;
                for (i=1;i<s1[0];i++)
                { //4
                    if (state==1)
                    { //5
                        if (s1[i]==B )
                        { //6
                            state=2;
                            prev=s1[i];
                            count=1;
                        } //6
                    } //5
                    else if (state==2)
                    { //6
                        if (s1[i]==prev ) count++;
                        else
                        { //7
                            if (count==1) status=MAYBE;
                            count=1;prev=s1[i];
                        } //7
                    } //6
                    if (status==MAYBE)
                    { //7
                        count=0; prev=s1[1]; decomposable=NO;
                        for (i=1;i<=s1[0];i++)
                        { //8
                            if (s1[i]==prev ) count++;
                            else if (count==1) decomposable = MAYBE;
                        } //8
                        if (count==1) decomposable = MAYBE;
                    } //7
                } //4
            } //3
        } //2
    } //1
    //5
    //4
}
/*if (status==NO)
{
    sprint(s1);
    cout<<" Pattern cannot be decomposable\n";
    cin>>z;
    if (z=='q') alldone=YES;
} */
if (status!=NO) status=check_string(s1);
return(status);
} //1

```

```

void print_decomposable()
{
    cout<<"in=";
    sprint(s1);
    cout<<" s=";
    charprint(s);
    cout<<" v1=";
    sprint(v1o);
    cout<<" v2=";
    sprint(v2o);
    cout<<" >> ";
    sprint(s2);
    cout<<" >> ";
    sprint(s3);
    cout<<"\n";
}

int variable_count(int x[100])
{
    int a=0,b=0,c=0,i,k;
    for (i=1;i<=x[0];i++)
        switch(x[i])
        {
            case 1: a=1;break;
            case 2: b=1;break;
            case 3: c=1;break;
            case 4: a=1;break;
            case 5: b=1;break;
            case 6: c=1;break;
        }
    return(a+b+c);
}

// VARIABLE HIGHSYMBOL FOR NOT CHECKING INVERSES

void increment_s1()
{
    int position=2,done=NO;

    if (decomposable_flag==YES)
    {
        store_all(s1);
        decomposable_count+=1.0;
        sum_of_squares+=transforms_per * transforms_per;
        transforms_per=0;
    }
    decomposable_flag=NO;
    while(done==NO)
    {
        if (s1[position]<HIGHSYMBOL)
        {
            s1[position]++;
            done=YES;
        }
        else
        {
            if (position==s1[0])

```

```

        {
            alldone=YES;
            done=YES;
        }
        else
        {
            s1[position]=1;
            position++;
        }
    }
    vhold=variable_count(s1);
    /* sprint(s1);
    cout<<"\n"; */
}

void int_copy(int x)
{
    s2[0]++;
    s3[0]++;
    s2[s2[0]]=x;
    s3[s3[0]]=x;
    /* to eliminate consecutive inverses */
    if (s3[0]>1 & abs(s3[s3[0]]-s3[s3[0]-1])==3)s3[0]-=2;
}

void string_copy(int x[10])
{
    int j,k;
    for (j=1;j<=x[0];j++)
    {
        s2[0]++;
        s3[0]++;
        s2[s2[0]]=x[j];
        s3[s3[0]]=x[j];
        /* to eliminate consecutive inverses */
        if (s3[0]>1 & abs(s3[s3[0]]-s3[s3[0]-1])==3)s3[0]-=2;
    }
}

void transform_s1()
{
    int i,v,x;

    s2[0]=0;
    s3[0]=0;

    for (i=1;i<=s1[0];i++)
    {
        x=s1[i];
        if (x==si) int_copy(si);
        else if (x==v1i) string_copy(v1io);
        else if (x==v2i) string_copy(v2io);
        else if (x==s) string_copy(so);
        else if (x==si) string_copy(sio);
        else if (x==v1) string_copy(v1o);
        else if (x==v2) string_copy(v2o);
    }
}

```

```

    }
    v=variable_count(s3);
    if (v<vhold)
    {
        print_decomposable();
        decomposable_flag=YES;
        transformation_count+=1.0;
        transforms_per+=1.0;
    }
    /*else
    {
        sprint(s1);
        cout<<">>";
        sprint(s3);
        cout<<" Not decomposable\n";
    } */
}

void setup_transformations() // if length is even
{
    //MUST PUT IN INVERSES TOO
    so[0]=1;so[1]=A;v1o[0]=2;v1o[1]=A_;v1o[2]=B;v2o[0]=1;v2o[1]=C;
    sio[0]=1;sio[2]=A_;v1io[0]=2;v1io[1]=B_;v1io[2]=A;v2io[0]=1;v2io[1]=C_;
    transform_s1();
    v2o[0]=2;v2o[1]=C;v2o[2]=A; v2io[0]=2; v2io[1]=A_; v2io[2]=C_;
    transform_s1();
    v1o[1]=B;v1o[2]=C_;v1io[1]=C;v1io[2]=B_;
    v2o[0]=1;v2o[1]=C;v2io[0]=1;v2io[1]=C_;
    transform_s1();
    v1o[1]=A_;v1o[2]=B; v1io[1]=B_;v1io[2]=A;
    v2o[0]=2;v2o[1]=C;v2o[2]=A_; v2io[0]=2; v2io[1]=A; v2io[2]=C_;
    transform_s1();
    so[0]=2;so[1]=A;so[2]=B_;sio[0]=2;sio[1]=B;sio[2]=A_;
    v1o[0]=1;v1o[1]=B;v1io[0]=1;v1io[1]=B_;
    v2o[0]=2;v2o[1]=B_;v2o[2]=C; v2io[0]=2;v2io[1]=C_;v2io[2]=B;
    transform_s1();
    so[0]=1;so[1]=A;v1o[0]=2;v1o[1]=A_;v1o[2]=B;v2o[0]=3;v2o[1]=A_;v2o[2]=C;
    sio[0]=1;sio[2]=A_;v1io[0]=2;v1io[1]=B_;v1io[2]=A;v2io[0]=3;v2io[1]=A_;v
    transform_s1();
}

void translate_s1()
{
    int i,r,hold[25];
    if (oddeven==0)
    {s=A;v1=B;v2=C;setup_transformations();}
    // otherwise, find how often the variables occur in row
    else
    {
        for (i=0;i<=25;i++) hold[i]=s1[i];
        r=rotate(s1,s1);
        if(r==0)
        {
            s=1;v1=2;v2=3;
            so[0]=1;so[1]=s;

```

```

        sio[0]=1;sio[1]=s+3;
        vlo[0]=2;vlo[1]=s+3;vlo[2]=v1;
        vlio[0]=2;vlio[1]=v1+3;vlio[2]=s;
        v2o[0]=1;v2o[1]=v2;
        v2io[0]=1;v2io[1]=v2+3;
        transform_s1();
    }
    for (i=0;i<=25;i++) s1[i]=hold[i];
}

void main()
{
    int endloop,i,x;

    clrscr();
    cout<<"ODD EVEN PROGRAM\n";
    cout<<"\n";
    cout<<"Press Ctrl+Print Screen now to have results printed.\n";
    cout<<"Please input the number of positions desired:";
    cin>>s1[0];
    oddeven = s1[0] % 2;
    // QUESTION ABOUT INVERSES

    for(i=1;i<=s1[0];i++) s1[i]= A;
    while (alldone==NO)
    {
        endloop=NO;
        while (endloop==NO)
        {
            increment_s1();
            //sprint(s1);cout<<'\\n';
            if (alldone==NO)
                x=ni_check_patterns();
            else endloop = YES;
            if (x==MAYBE) endloop=YES;
        }
        //sprint(s1);cout<<"\\n";
        translate_s1();
    }
    cout<<"Number of decomposable strings=";
    cout<<decomposable_count;
    cout<<"\\n";
    cout<<"Total transformations resulting in decomposition=";
    cout<<transformation_count;
    cout<<"\\n";
    cout<<"Transformations / Strings=";
    cout<<transformation_count/decomposable_count,"\\n";
    cout<<"\\n";
    cout<<"Estimated variance of above=";
    cout<< (sum_of_squares - ((transformation_count * transformation_count )
        ) / decomposable_count;
    cout<<"\\n";
}

```

Examples of output from the previous program.

marvinoe
'ODD EVEN PROGRAM

Press Ctrl+Print Screen now to have results printed.
Please input the number of positions desired:3
in=ABC s=A v1=A*B v2=C >> AA*BC >> BC
Number of decomposable strings=1
Total transformations resulting in decomposition=1
Transformations / Strings=1
Estimated variance of above=0

Type 'EXIT' to return to program
A:\>marvinoe
'ODD EVEN PROGRAM

Press Ctrl+Print Screen now to have results printed.
Please input the number of positions desired:4
in=ABAB s=A v1=A*B v2=C >> AA*BAA*B >> BB
in=ABAB s=A v1=A*B v2=CA >> AA*BAA*B >> BB
in=ABAB s=A v1=A*B v2=CA* >> AA*BAA*B >> BB
in=ABAB s=A v1=B v2=B*C >> AB*BAB*B >> AA
in=ABAB s=A v1=A*B v2=A*CA >> AA*BAA*B >> BB
in=ABCB s=A v1=A*B v2=CA >> AA*BCAA*B >> BCB
in=AABC s=A v1=BC* v2=C >> AABC*C >> AAB
Number of decomposable strings=3
Total transformations resulting in decomposition=7
Transformations / Strings=2.333333
Estimated variance of above=3.555556

Type 'EXIT' to return to program
A:\>marvinoe
'ODD EVEN PROGRAM

Press Ctrl+Print Screen now to have results printed.
Please input the number of positions desired:5
in=ABCAB s=A v1=A*B v2=C >> AA*BCAA*B >> BCB
in=ABCCC s=A v1=A*B v2=C >> AA*BCCC >> BCCC
Number of decomposable strings=2
Total transformations resulting in decomposition=2
Transformations / Strings=1
Estimated variance of above=0

Type 'EXIT' to return to program
A:\>marvinoe
'ODD EVEN PROGRAM

Press Ctrl+Print Screen now to have results printed.
Please input the number of positions desired:6
in=ABCAAB s=A v1=A*B v2=CA* >> AA*BCA*AAA*B >> BCB
in=ABABAB s=A v1=A*B v2=C >> AA*BAA*BAA*B >> BBB
in=ABABAB s=A v1=A*B v2=CA >> AA*BAA*BAA*B >> BBB
in=ABABAB s=A v1=A*B v2=CA* >> AA*BAA*BAA*B >> BBB
in=ABABAB s=A v1=B v2=B*C >> AB*BAB*BAB*B >> AAA
in=ABABAB s=A v1=A*B v2=A*CA >> AA*BAA*BAA*B >> BBB
in=ABCBAB s=A v1=A*B v2=CA >> AA*BCAA*BAA*B >> BCBB
in=ABBCAB s=A v1=B v2=B*C >> AB*BBB*CAE*B >> ACA
in=ABACCB s=A v1=A*B v2=A*CA >> AA*BAA*CAA*CAA*B >> BCCB
in=AAAABC s=A v1=BC* v2=C >> AAAABC*C >> AAAAB
in=ABCABC s=A v1=A*B v2=C >> AA*BCAA*BC >> BCBC
in=ABCABC s=A v1=BC* v2=C >> ABC*ABC*C >> ABAB

Number of decomposable strings=7
Total transformations resulting in decomposition=12
Transformations / Strings=1.714286
Estimated variance of above=1.918367

Type 'EXIT' to return to program
A:\>marvin

The following C++ program uses the improved version of the Baumslag-Taylor algorithm with the input being the length of the relator.

```

//MARVIN8 - cuts out most repetitions uses additional criteria for
//determining a trivial center.
#include <constrea.h>
#include <iostream.h>

int count2=0,count4=0,exponent[20],hi,low,n;
float count1=0.0,count3=0.0;

class element{
public:
    element *lptr;
    int x;
    int last;
    int active;
    int id;
    int boundary;
    element *rptr;

    //procedures
    element(element *prev, int k)
    //create element with value linked to prev.
    { lptr=prev; // links to previous element
      x=k;       // inserts value
      last=0;
      active = 1;
      boundary=0;
      rptr=NULL;
    };
    element() { lptr=NULL;x=-2;rptr=NULL;active=1; };
    void eleprint();
    int increment();
};

class list // circular list
{
public:
    element *p;
    int count;

    //procedures
    void print();
    list(void);
    int a_test();
    int cancel();
    element* remove(element *r);
    void find_pattern(int print);
    void revive_all();
    int any_a();
    void delete_as_before(element *a, int k);
    void delete_as_after(element *a, int k);
    void write_bs(element *p, int k, int print);
    int number(int k);
    int highest();
    void exp_inc();
    void reverse();
};

element* next(element *p)
{
    element *q;
    int i=0;

```

```

        q=p->rptra;
        while(!q->active & i <= n)
        {q=q->rptra; i++;}
        return(q);
    }
    element* prev(element *p)
    {
        element *q;
        int i=0;
        q=p->lptra;
        while(!q->active & i <= n)
        {q=q->lptra; i++;}
        return(q);
    }
    list:: list(void) // creates initial list
    {
        element *q,r;
        int i;
        p= new element(NULL,-2); // p is defined as the list pointer in l
        q=p;
        p->id=1;
        for (i=1;i<n;i++)
        {
            q->rptra= new element(q,-2);
            q->rptra->id=i+1;
            q=q->rptra;
        }
        q->rptra=p; // link last element to first
        p->lptra=q;
        q->last = 1;
        count=n;
    }

    void list:: print() // prints list of elements
    {
        element *q;
        int i;

        q=p;
        for (i=1;i<=count;i++)
        {
            q->eleprint(); // to print one element
            q=q->rptra;
        }
        cout << " ";
    }

    void element::eleprint() // prints one element
    {
        switch (x)
        {
            case -2: cout << "B*";break;
            case -1: cout << "A*"; break;
            case 1:  cout << "A"; break;
            case 2:  cout << "B"; break;
        }
    }
}

```

```

element* list::remove(element *r) //removes element from list
{
    element *q;    // returns next element in list
    r->active = 0;
    count--;
    if (count > 0) q=next(r);
    return(q);
}

int list::cancel()
{
    element *q;
    int c=0,i;

    i=1;
    q=p;    //address of first element on list
    while (i <= n & count >=2 )    // string is in circular format
    {
        //next=q->rpitr; //the next element in the list
        if (q->x + q->rpitr->x == 0) // checks two adjacent elements
        {
            c=1;
            q=remove(q);
            q=remove(q);
            i=0;    // string must be rechecked
        }
        else
        {
            //          cout << "Element passed over";
            if (count >= 2) q=next(q);
            i++;
        }
    }
    return(c);
}

int element::increment() // increments the counter returns 1 if all done
{
    int done=0;
    switch (x)
    {
        case -2: {x=-1;break;}
        case -1: {x=1; break;}    //goes from A* to A
        case 1:  {x=2; break;}
        case 2:  {x = -2;
                    if (last) done = 1;    // if last one
                    else done = rpitr->increment();    // carry to nxt digit
                    break;
                }
    }
    return (done);
}

int list::a_test() //tests that the numbers of a's and a*'s are the same
{
    element *q;
    int i,a1=0,a2=0;
    q=p;

```

```

        for(i=1;i<=count;i++)
        {
            if (q->x== -1) a1=a1+1;
            if (q->x== 1) a2=a2+1;
            q=q->rptra;
        }
        if (a1==a2 & a1!=0 ) return(1);
        else return (0);
    }

int abs(int x)
{ if (x>=0) return(x); else return(-x); }

int min(int x, int y)
{ if (x>y) return(y); else return(x); }

int list::any_a()
{
    element *q;
    int i=0;
    q=next(p);          //finds first live element
    while (i<count & abs(q->x) != 1)
        (i++; q=next(q));
    if (abs(q->x)==1) i=1; else i=0;
    return(i);
}

element* next_b(element *q)
{
    element *b;
    b=next(q);
    while(abs(b->x)!=2) b=next(b);
    return(b);
}

element* prev_a(element *b)
{
    element *a;
    a=b;
    while (abs(a->x) != 1) a=prev(a);
    return(a);
}

element* next_a(element *b)
{
    element *a;
    a=b;
    while (abs(a->x) != 1) a=next(a);
    return(a);
}

int count_back(element *a)
{
    int i=0;          element *q;
    q=a;
    while (abs(q->x)==1) {i+=q->x; q=prev(q);}
    return(i);
}

```

```

    )
}

void list::find_pattern(int print)    // called only if count >= 3
{
    element *a,*b,*c,*q;
    int c1=0,c3=0,finished=0,k;
    if(any_a())
    {
        c=p; finished=0;
        while(!finished)
        {
            b=c;    // to look for the next group of B's
            b=next_b(b);    //if there is an A there must be a B
            a=prev_a(b);
            c=next_a(b);
            if (a->x + c->x == 0) finished = 1;
            q=a;
            while (q!=c)
            {
                if (q->boundary==1) finished = 0;    // bound
                q=q->rptr;    //we are checking deleted elemen
            }
        }
        c1=count_back(a);
        c3=count_forward(c);
        if (min(abs(c1),abs(c3)) == abs(c1))
            k= -c1; else k=c3;
        //b=find_first(b);    //MUST FIND FIRST B BEFORE DELETING A'S
        b=a;
        while(abs(b->x)==1) b=next(b);
        delete_as_before(a,k);
        write_bs(b,k,print);
        delete_as_after(c,k);
        find_pattern(print);
    }
    else
    {
        q=next(p);
        while(count>0)
        {
            //cout << q->x;
            if (print)
            {
                q->eleprint();
                cout << "0 ";
            }
            q=remove(q);
            push_exp(0);
        }
    }
}

int list::number(int k)
{
    int total=0,power=1,i;
    element *q;
    q=p;
    for(i=1;i<=k;i++)q=q->rptr;
}

```

```

int count_forward(element *a)
{
    int i=0;          element *q;
    q=a;
    while (abs(q->x)==1) {i+=q->x; q=next(q);}
    return(i);
}

void list::delete_as_before(element *a, int k)
{
    int i;
    element *q;
    q=a;
    for(i=1;i<=abs(k);i++)
    {
        q=remove(q);
        q=prev(q);
    }
}

void list::delete_as_after(element *a, int k)
{
    int i;
    element *q;
    q=a;
    for(i=1;i<=abs(k);i++) q=remove(q); //removes and goes to next
}

element* find_first(element *p)
{
    int x;
    element *q;

    x=p->x;
    q=p;
    while (abs(q->x) == x) q=prev(q);
    if (q!=p) q=next(q);
    return(q);
}

void push_exp(int e)
{ exponent[0]++;exponent[exponent[0]]=e;}

void list::write_bs(element *p, int k, int print)
{
    element *q,*r;
    int i;
    q=p;
    while (abs(q->x)==2)
    {
        if (print)
        {
            q->eleprint();
            cout << k << " ";
        }
        r=p; for (i=1;i<=n;i++)
            {r->boundary=0; r=r->rptra;} //removes old barrier
        q->boundary=1;
        q=remove(q);
        push_exp(k);
    }
}

```

```

        for(i=1;i<=count;i++)
        {
            switch(q->x)
            {
                case -2: total+=power*1;break;
                case -1: total+=power*2;break;
                case 1:  total+=power*3;break;
                case 2:  total+=power*4;break;
            }
            q=q->rptra;
            power*=5;
        }
        return(total);
    }

int list::highest()    //HIGHEST FORM OF STRING TEST
{
    int i,x,y,z=1;
    x=number(0);
    for(i=0;i<=count-1;i++)
    {
        y=number(i);
        if (y>x) z=0;
    }
    return(z);
}

void list::revive_all()
{
    int i;
    element *q;

    q=p;
    for(i=1;i<=n;i++)
    {
        q->active = 1; q->boundary=0; q=q->rptra;
    }
    count=n;
}

void trivial_center()
{
    count4++;
    cout<<" [TRIVIAL CENTER]  ";           // trivial center
}

void list::reverse()
{
    element *q;
    int i;
    q=p;
    for(i=1;i<=n;i++)
    {
        q->x*= -1;
        q=q->rptra;
    }
}

int hi_and_low_occur_once()
{
    int i,j=0,k=0,sum;
    hi=low=exponent[1];
}

```

```

    for (i=1;i<=exponent[0];i++)
    {
        sum+=exponent[i];
        if(exponent[i]>hi) hi = exponent[i];
        if(exponent[i]<low) low=exponent[i];
    }
    for (i=1;i<=exponent[0];i++)
    {
        if (exponent[i]==hi) j++;
        if (exponent[i]==low) k++;
    }
    if (j==1 & k==1) return(1); else return(0);
}

void list::exp_inc()
{
    int sum;
    if (hi_and_low_occur_once())    count1+=1.0;    // finitely generated
    else { count2++;
        if ((hi-low)>1 | sum !=0) trivial_center();
        else
        {
            exponent[0]=0; //zero out exponent list
            reverse(); //reverse string
            find_pattern(0); //check pattern without
            reverse(); //reverse string
            if (!hi_and_low_occur_once()) trivial_center();
            //check highest and lowest subscripts of A
        }
    }
    count3+=1.0;
}

int main(void)
{
    int i=0,finished=0;

    clrscr();
    cout << "MARVIN8 " << endl;
    cout << "Turn on Ctrl + Print Screen to print results." << endl;
    cout << "What length string do you want to use?";
    cin >> n; //input string length
    list l; //creates working initial string of length n
    while (finished == 0)
    {
        i++;
        l.print();
        cout << endl;cout << i << endl;
        if (l.a_test()) //tests number of A's correct
            if (!l.cancel() & l.highest()) //remove contiguous a and
            {
                exponent[0]=0;
                l.print();
                l.find_pattern(1);
                l.exp_inc();
                cout << endl;
            }
        finished = l.p->increment(); // to get to next string
    }
}

```

```

        l.revive_all();
    }
    cout << "** WARNING: THESE NUMBERS MAY NOT BE ACCURATE FOR SOME STRING L
    cout << "          Use program 4B for accurate numbers." << endl;
    cout << "Normal closure of B is finitely generated: " << count1 << endl;
    cout << "Normal closure of B NOT finitely generated: "<< count2 << endl;
    if (count3!=0)
    cout << "Normal closure of B finitely generated in " << setw(6) << ((cou
    cout << "Trivial Center: "<< count4 << endl;
    return(0);
}

```

Examples of output from the previous program.

```

8
MARVIN8
Turn on Ctrl + Print Screen to print results.
What length string do you want to use?4
B*A*B*A      B*1 B*0
AB*A*B       B*-1 B0
A*B*AB       B*1 B0
A*BAB        B1 B0
** WARNING: THESE NUMBERS MAY NOT BE ACCURATE FOR SOME STRING LENGTHS **
    Use program 4B for accurate numbers.
Normal closure of B is finitely generated: 4
Normal closure of B NOT finitely generated: 0
Normal closure of B finitely generated in 100% of cases.
Trivial Center: 0

Type 'EXIT' to return to program
A:\>marvin8
MARVIN8
Turn on Ctrl + Print Screen to print results.
What length string do you want to use?5
B*A*B*B*A      B*1 B*1 B*0 [TRIVIAL CENTER]
B*B*A*B*A      B*-1 B*-1 B*0 [TRIVIAL CENTER]
AB*B*A*B       B*-1 B*-1 B0 [TRIVIAL CENTER]
A*B*B*AB       B*1 B*1 B0 [TRIVIAL CENTER]
AB*A*BB        B*-1 B0 B0 [TRIVIAL CENTER]
ABA*BB         B-1 B0 B0 [TRIVIAL CENTER]
A*B*ABB        B*1 B0 B0 [TRIVIAL CENTER]
A*BABB         B1 B0 B0 [TRIVIAL CENTER]
** WARNING: THESE NUMBERS MAY NOT BE ACCURATE FOR SOME STRING LENGTHS **
    Use program 4B for accurate numbers.
Normal closure of B is finitely generated: 0
Normal closure of B NOT finitely generated: 8
Normal closure of B finitely generated in 0% of cases.
Trivial Center: 8

Type 'EXIT' to return to program
A:\>marvin8
MARVIN8
Turn on Ctrl + Print Screen to print results.
What length string do you want to use?6
B*A*B*B*B*A      B*1 B*1 B*1 B*0 [TRIVIAL CENTER]
B*B*A*B*B*A      B*-1 B*-1 B*0 B*0 [TRIVIAL CENTER]
B*B*B*A*B*A      B*-1 B*-1 B*-1 B*0 [TRIVIAL CENTER]
B*A*A*B*AA       B*2 B*0
AB*B*B*A*B       B*-1 B*-1 B*-1 B0 [TRIVIAL CENTER]
AAB*A*A*B        B*-2 B0
A*B*B*B*AB       B*1 B*1 B*1 B0 [TRIVIAL CENTER]
A*A*B*AAB        B*2 B0
A*A*BAAB         B2 B0
AB*B*A*BB        B*-1 B*-1 B0 B0 [TRIVIAL CENTER]
A*B*B*ABB        B*1 B*1 B0 B0 [TRIVIAL CENTER]
A*BBABB          B1 B1 B0 B0 [TRIVIAL CENTER]
AB*A*BBB         B*-1 B0 B0 B0 [TRIVIAL CENTER]
ABA*BBB          B-1 B0 B0 B0 [TRIVIAL CENTER]
A*B*ABBB         B*1 B0 B0 B0 [TRIVIAL CENTER]
A*BABBB          B1 B0 B0 B0 [TRIVIAL CENTER]
** WARNING: THESE NUMBERS MAY NOT BE ACCURATE FOR SOME STRING LENGTHS **
    Use program 4B for accurate numbers.
Normal closure of B is finitely generated: 4
Normal closure of B NOT finitely generated: 12

```

Normal closure of B finitely generated in 25% of cases.
Trivial Center: 12

Type 'EXIT' to return to program

A:\>marvin8

MARVIN8

Turn on Ctrl + Print Screen to print results.

What length string do you want to use??

```

B*ABA*AB*      B-2 B*0 B*0 [TRIVIAL CENTER]
ABABA*AB*      B-1 B*1 B0
AABBA*AB*      B-2 B-2 B*0 [TRIVIAL CENTER]
AABA*BA*AB*    B-1 B*1 B0
B*B*B*ABA*AB*  B*1 B*1 B*1 B*1 B0 [TRIVIAL CENTER]
A*B*ABA*AB*    B*1 B-1 B*0
A*BAABA*AB*    B1 B-1 B*0
B*B*ABBA*AB*   B*1 B*1 B*1 B0 B0 [TRIVIAL CENTER]
B*ABBEA*AB*    B-1 B-1 B-1 B*0 B*0 [TRIVIAL CENTER]
ABBBBA*AB*     B-1 B-1 B-1 B-1 B*0 [TRIVIAL CENTER]
A*B*B*B*AB*AB* B*1 B*1 B*1 B*1 B*0 [TRIVIAL CENTER]
B*A*B*B*B*AB*  B*1 B*1 B*1 B*0 B*0 [TRIVIAL CENTER]
B*B*A*B*B*AB*  B*-1 B*-1 B*-1 B*0 B*0 [TRIVIAL CENTER]
B*B*B*A*B*AB*  B*-1 B*-1 B*-1 B*-1 B*0 [TRIVIAL CENTER]
ABA*AB*AB*AB*  B-1 B*1 B*0
A*A*B*AB*AB*   B*1 B*-1 B*0
A*A*B*B*AB*AB* B*2 B*2 B*0 [TRIVIAL CENTER]
A*B*A*B*AB*AB* B*1 B*-1 B*0
B*A*A*B*AB*AB* B*2 B*0 B*0 [TRIVIAL CENTER]
A*BA*B*AB*AB*  B*1 B*-1 B0
A*B*A*BAAB*AB* B1 B*-1 B*0
B*A*A*BAAB*AB* B2 B*0 B*0 [TRIVIAL CENTER]
A*BA*BAAB*AB*  B1 B*-1 B0
A*A*BBAAB*AB*  B2 B2 B*0 [TRIVIAL CENTER]
B*B*B*A*BAB*AB B*-1 B*-1 B*-1 B*-1 B0 [TRIVIAL CENTER]
AB*A*A*BAB*AB* B*-1 B1 B*0
ABA*A*BAB*AB*  B-1 B1 B*0
A*A*B*AB*AB*AB B*1 B*-1 B0
A*A*BAB*AB*AB* B1 B*-1 B0
B*B*A*BB*AB*AB B*-1 B*-1 B*-1 B0 B0 [TRIVIAL CENTER]
B*A*BBB*AB*AB* B1 B1 B1 B*0 B*0 [TRIVIAL CENTER]
A*BBBB*AB*AB*  B1 B1 B1 B1 B*0 [TRIVIAL CENTER]
A*BA*BAAB*AB*  B1 B-1 B0
A*A*BAB*AB*AB* B1 B-1 B0
AABA*A*BB*AB*  B-2 B0 B0 [TRIVIAL CENTER]
A*A*BAAB*AB*AB B2 B0 B0 [TRIVIAL CENTER]
ABBA*BBB*AB*AB B-1 B-1 B0 B0 B0 [TRIVIAL CENTER]
A*BBABBB*AB*AB B1 B1 B0 B0 B0 [TRIVIAL CENTER]
ABA*BBBB*AB*AB B-1 B0 B0 B0 B0 [TRIVIAL CENTER]
A*BABBBB*AB*AB B1 B0 B0 B0 B0 [TRIVIAL CENTER]

```

** WARNING: THESE NUMBERS MAY NOT BE ACCURATE FOR SOME STRING LENGTHS **

Use program 4B for accurate numbers.

Normal closure of B is finitely generated: 16

Normal closure of B NOT finitely generated: 24

Normal closure of B finitely generated in 40% of cases.
Trivial Center: 24

Type 'EXIT' to return to program

A:\>

The following C++ program is a more accurate version of the previous program since it does not remove isomorphic strings from consideration. Due to this fact it runs much slower. The difference in accuracy from the other program is minor.

```

//MARVIN4B - counts of finitely generated for every possible string.
#include <constrea.h>
#include <iostream.h>

int count2=0,count4=0,exponent[20],n;
float count1=0.0,count3=0.0;

class element(
    public:
        element *lptr;
        int x;
        int last;
        int active;
        int boundary;
        int id;
        element *rptr;

        //procedures
        element(element *prev, int k)
        //create element with value linked to prev.
        { lptr=prev; // links to previous element
          x=k; // inserts value
          last=0;
          active = 1;
          boundary=0;
          rptr=NULL;
        };
        element() { lptr=NULL;x=-2;rptr=NULL;active=1; };
        void eleprint();
        int increment();
    );

class list // circular list
{
    public:
        element *p;
        int count;

        //procedures
        void print();
        list(void);
        int a_test();
        int cancel();
        element* remove(element *r);
        void find_pattern();
        void revive_all();
        int any_a();
        void delete_as_before(element *a, int k);
        void delete_as_after(element *a, int k);
        void write_bs(element *p, int k);
};

element* next(element *p)
{
    element *q;
    int i=0;
    q=p->rptr;
    while(!q->active & i <= n)
        (q=q->rptr; i++;)
    return(q);
}

```

```

    }
    element* prev(element *p)
    {
        element *q;
        int i=0;
        q=p->lptr;
        while(!q->active & i <= n)
            (q=q->lptr; i++);
        return(q);
    }

list:: list(void) // creates initial list
{
    element *q,r;
    int i;
    p= new element(NULL,-2); // p is defined as the list pointer in l
    q=p;
    p->id=1;
    for (i=1;i<n;i++)
    {
        q->rpitr= new element(q,-2);
        q->rpitr->id=i+1;
        q=q->rpitr;
    }
    q->rpitr=p; // link last element to first
    p->lptr=q;
    q->last = 1;
    count=n;
}

void list:: print() // prints list of elements
{
    element *q;
    int i;

    q=p;
    for (i=1;i<=count;i++)
    {
        q->eleprint(); // to print one element
        q=q->rpitr;
    }
    cout << " ";
}

void element::eleprint() // prints one element
{
    switch (x)
    {
        case -2: cout << "B*";break;
        case -1: cout << "A*"; break;
        case 1:  cout << "A"; break;
        case 2:  cout << "B"; break;
    }
}

element* list::remove(element *r) //removes element from list
{
    element *q; // returns next element in list
    r->active = 0;
}

```

```

        count--;
        if (count > 0) q=next(r);
        return(q);
    }

int list::cancel()
{
    element *q;
    int c=0,i;

    i=1;
    q=p; //address of first element on list
    while (i <= n & count >=2 ) // string is in circular format
    {
        //next=q->rptra; //the next element in the list
        if (q->x + q->rptra->x == 0) // checks two adjacent elements
        {
            c=1;
            q=remove(q);
            q=remove(q);
            i=0; // string must be rechecked
        }
        else
        {
            // cout << "Element passed over";
            if (count >= 2) q=next(q);
            i++;
        }
    }
    return(c);
}

int element::increment() // increments the counter returns 1 if all done
{
    int done=0;
    switch (x)
    {
        case -2: {x=-1;break;}
        case -1: {x=1; break;} //goes from A* to A
        case 1: {x=2; break;}
        case 2: {x = -2;
                if (last) done = 1; // if last one
                else done = rptra->increment(); // carry to nxt digit
                break;
            }
    }
    return (done);
}

int list::a_test() //tests that the numbers of a's and a*'s are the same
{
    element *q;
    int i,a1=0,a2=0;
    q=p;
    for(i=1;i<=count;i++)
    {
        if (q->x== -1) a1=a1+1;
        if (q->x== 1) a2=a2+1;
        q=q->rptra;
    }
}

```

```

    }
    if (a1==a2) return(1);
    else return (0);
}

int abs(int x)
{ if (x>=0) return(x); else return(-x); }

int min(int x, int y)
{ if (x>y) return(y); else return(x); }

int list::any_a()
{
    element *q;
    int i;
    q=next(p);      //finds first live element
    while(i<count & abs(q->x) != 1)
        (i++; q=next(q));
    if (abs(q->x)==1) i=1; else i=0;
    return(i);
}

element* next_b(element *q)
{
    element *b;
    b=next(q);
    while(abs(b->x)!=2) b=next(b);
    return(b);
}

element* prev_a(element *b)
{
    element *a;
    a=b;
    while (abs(a->x) != 1) a=prev(a);
    return(a);
}

element* next_a(element *b)
{
    element *a;
    a=b;
    while (abs(a->x) != 1) a=next(a);
    return(a);
}

int count_back(element *a)
{
    int i=0;      element *q;
    q=a;
    while (abs(q->x)==1) (i+=q->x; q=prev(q));
    return(i);
}

int count_forward(element *a)
{
    int i=0;      element *q;
    q=a;

```

```

        while (abs(q->x)==1) {i+=q->x; q=next(q);}
        return(i);
    }

void list::delete_as_before(element *a, int k)
{
    int i;
    element *q;
    q=a;
    for(i=1;i<=abs(k);i++)
    {
        q=remove(q);
        q=prev(q);
    }
}

void list::delete_as_after(element *a, int k)
{
    int i;
    element *q;
    q=a;
    for(i=1;i<=abs(k);i++) q=remove(q); //removes and goes to next
}

element* find_first(element *p)
{
    int x;
    element *q;

    x=p->x;
    q=p;
    while (abs(q->x) == x) q=prev(q);
    if (q!=p) q=next(q);
    return(q);
}

void push_exp(int e)
{ exponent[0]++;exponent[exponent[0]]=e;}

void list::write_bs(element *p, int k)
{
    element *q;
    q=p;
    while (abs(q->x)==2)
    {
        //q->eleprint();
        //cout << k << " ";
        q->boundary=1;
        q=remove(q);
        push_exp(k);
    }
}

void list::find_pattern() // called only if count >= 3
{
    element *a,*b,*c,*q;
    int c1=0,c3=0,finished=0,k;
    if(any_a())
    {
        c=p; finished=0;
        while(!finished)

```

```

    {
        b=c;          // to look for the next group of B's
        b=next_b(b);   //if there is an A there must be a B
        a=prev_a(b);
        c=next_a(b);
        if (a->x + c->x == 0) finished = 1;
        q=a;
        while (q!=c)
        {
            if (q->boundary==1) finished = 0;          // bound
            q=q->rptra;          //we are checking deleted elemen
        }
        c1=count_back(a);
        c3=count_forward(c);
        if (min(abs(c1),abs(c3)) == abs(c1))
            k= -c1; else k=c3;
        //b=find_first(b);          //MUST FIND FIRST B BEFORE DELETING A'S
        b=a;
        while(abs(b->x)==1) b=next(b);
        delete_as_before(a,k);
        write_bs(b,k);
        delete_as_after(c,k);
        find_pattern();
    }
    else
    {
        q=next(p);
        while(count>0)
        {
            //cout << q->x;
            //q->eleprint();
            //cout << "0 ";
            q=remove(q);
            push_exp(0);
        }
    }
}

void list::revive_all()
{
    int i;
    element *q;

    q=p;
    for(i=1;i<=n;i++)
    {
        q->active = 1; q->boundary=0; q=q->rptra;
    }
    count=n;
}

void exp_inc()
{
    int i,j=0,k=0,hi,low;
    hi=low=exponent[1];
    for (i=1;i<=exponent[0];i++)
    {
        if(exponent[i]>hi) hi = exponent[i];
        if(exponent[i]<low) low=exponent[i];
    }
}

```

```

    }
    for (i=1;i<=exponent[0];i++)
    {
        if (exponent[i]==hi) j++;
        if (exponent[i]==low) k++;
    }
    if (j==1 & k==1) count1+=1.0;
    else {
        count2++;
        if ((hi - low) > 1) count4 ++;
    }
    count3+=1.0;
}

int main(void)
{
    int i=0,finished=0;

    clrscr();
    cout << "Turn on Ctrl + Print Screen to print results." << endl;
    cout << "What length string do you want to use?";
    cin >> n; //input string length
    cout << endl;
    cout << "Please wait for totals to be printed." << endl;
    list l; //creates working initial string of length n
    while (finished == 0)
    {
        i++;
        l.print();
        cout << endl;cout << i << endl;
        if (l.a_test()) //tests number of A's correct
            if (!l.cancel()) //remove contiguous a and a*
            {
                exponent[0]=0;
                //l.print();
                l.find_pattern();
                //cout << endl;
                exp_inc();
            }
        finished = l.p->increment(); // to get to next string
        l.revive_all();
    }
    count2-=2; // for strings that are all Bs or all B*.
    cout << "Normal closure of B is finitely generated: " << count1 << endl;
    cout << "Normal closure of B NOT finitely generated: "<< count2 << endl;
    if (count3!=0)
    cout << "Normal closure of B finitely generated in " << setw(6) << ((cou
    cout << "Trivial Center: " << count4 << endl;
    return(0);
}

```

Examples of output from the previous program.

```

marvin4b
Turn on Ctrl + Print Screen to print results.
What length string do you want to use?4

Please wait for totals to be printed.
Normal closure of B is finitely generated: 16
Normal closure of B NOT finitely generated: 0
Normal closure of B finitely generated in 88.888885% of cases.
Trivial Center: 0

Type 'EXIT' to return to program
A:\>marvin4b
Turn on Ctrl + Print Screen to print results.
What length string do you want to use?5

Please wait for totals to be printed.
Normal closure of B is finitely generated: 0
Normal closure of B NOT finitely generated: 40
Normal closure of B finitely generated in 0% of cases.
Trivial Center: 0

Type 'EXIT' to return to program
A:\>marvin4b
Turn on Ctrl + Print Screen to print results.
What length string do you want to use?6

Please wait for totals to be printed.
Normal closure of B is finitely generated: 24
Normal closure of B NOT finitely generated: 72
Normal closure of B finitely generated in 24.489796% of cases.
Trivial Center: 0

Type 'EXIT' to return to program
A:\>marvin4b
Turn on Ctrl + Print Screen to print results.
What length string do you want to use?7

Please wait for totals to be printed.
Normal closure of B is finitely generated: 112
Normal closure of B NOT finitely generated: 168
Normal closure of B finitely generated in 39.716312% of cases.
Trivial Center: 56

Type 'EXIT' to return to program
A:\>marvin4b
Turn on Ctrl + Print Screen to print results.
What length string do you want to use?8

Please wait for totals to be printed.
Normal closure of B is finitely generated: 288
Normal closure of B NOT finitely generated: 576
Normal closure of B finitely generated in 33.256351% of cases.
Trivial Center: 352

Type 'EXIT' to return to program
A:\>marvin4b
Turn on Ctrl + Print Screen to print results.
What length string do you want to use?9

Please wait for totals to be printed.

```

Normal closure of B is finitely generated: 720
Normal closure of B NOT finitely generated: 1728
Normal closure of B finitely generated in 29.387754% of cases.
Trivial Center: 1224

Type 'EXIT' to return to program
A:\>

Bibliography

- [1] Magnus, Wilhelm, 1930, Über discontinuierliche Gruppen mit einer definierenden Relation (Der Freiheitssatz), *J. f. d. reine u. angew. Math.* **163**, 141-165.
- [2] Shenitzer, Abe, 1955, Decomposition of a Group with a Single Defining Relator into a Free Product, *Proceedings of the American Mathematical Society* **6**, 273-279.
- [3] Whitehead, J. H. C., 1936, On equivalent sets of elements in a free group, *Annals of Math.* **37**, 782-800.
- [4] Shenitzer, Abe, 1995, In Memory of My Friend Wilhelm Magnus, *The Mathematical Intelligencer* **17** No. 2, 63-64.
- [5] Grushko, I., 1940, Über die Basen eines freien Produkts von Gruppen, *Rec. Math. (Mat. Sbornik) N. S.* **8** (50), 169-182 (Russian, German Summary).
- [6] Murasugi, Kunio, 1964, The Center of a Group with a Single Defining Relation, *Annals of Mathematics* **155**, 246-251.
- [7] Baumslag, Gilbert and Taylor, Tekla, 1968, The Center Of a Group with One Defining Relator, *Annals of Mathematics* **175**, 315-319.
- [8] Levin, Frank, 1994, Testing for the Center of a One Relator Group, *The Mathematical Legacy of Wilhelm Magnus: Groups, Geometry and Special Functions*, Publ. American Mathematical Society, (Providence, R.I.).
- [9] Lothaire, M, Encyclopedia of Mathematics, Vol. 17, Combinatorics on Words. Pg. 18-31, Publ. Addison-Wesley Publishing Company, 1983.
- [10] Aho, A.V. and M.J. Corasick, 1975, Efficient string matching: an aid to bibliographic search, *Comm. ACM* **18**(6), 333-340.
- [11] Commentz-Walter, B., A string matching algorithm fast on the average, in: ed., *Proc. 6th Internat. Coll. on Automata, Languages and Programming* (Springer, Berlin, 1979) 118-132.