2005

The Necklace Illustrator Web Applet

David W. Fraser

The Necklace Illustrator Web Applet

by

David W. Fraser

A Report Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
Master of Science
in
Applied Mathematics

Approved by

Principal Advisor: Paul Wilson
Dr. Paul Wilson

Committee Member: David Hart
Prof. David Hart

Committee Member: Darren Narayan
Dr. Darren Narayan

Committee Member: Hossein Shahmahamad
Dr. Hossein Shahmohamad

Department of Mathematics and Statistics
Rochester Institute of Technology
May, 2005

RIT DML Electronic Thesis & Dissertation (ETD)
Thesis/Capstone Project
Author Permission Statement

Title of thesis/Capstone project:
The Necklace Illustrator Web Applet

I, David Fraser, hereby grant the nonexclusive license to the Rochester Institute of Technology Digital Media Library (RIT DML) to archive and provide electronic access to my thesis/Capstone project in perpetuity.

I hereby certify that, if appropriate, I have obtained and attached written permission statements from the owners of each third party copyrighted matter to be included in my thesis/Capstone project. I certify that the version I submitted is the same as that approved by my advisor and/or committee.

I hereby grant to the Rochester Institute of Technology and its agents the non-exclusive license to archive and make accessible my thesis/Capstone project in whole or in part in all forms of media. I understand that my work, in addition to its bibliographic record and abstract, will be available to the world-wide community of scholars and researchers through the RIT DML.

I retain all other ownership rights to the copyright of my thesis/Capstone project. I also retain the right to use in future works (such as articles or books) all or part of my thesis/Capstone project. The Rochester Institute of Technology does not require registration of copyright for thesis/Capstone projects

Signature of author: David Fraser
Date: 8/31/05 Degree: MS
College: Science
Program: Applied Mathematics

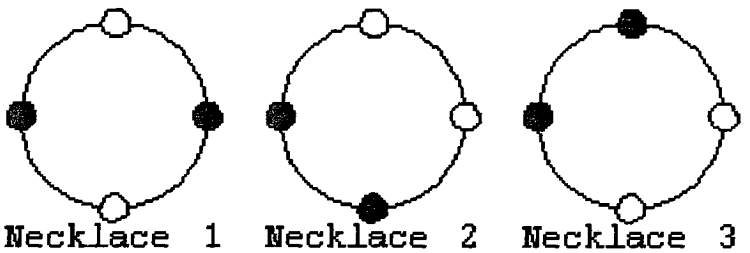
Table of Contents

Section I: Background

Introduction to Necklaces

In terms of Combinatorics and Graph Theory, a necklace is a cycle with each vertex (or "bead") given a certain color. The problem is to count the number of necklaces having n beads if k colors are available. The number of distinct necklaces can be determined by building the pattern inventory, a polynomial with variables representing different colors, the exponents of the variables representing the number of beads of that color, and the coefficients representing the number of distinct necklaces with that coloring. For example, a necklace made of five beads using the colors black and white has the pattern inventory $b^5 + 2b^3w^2 + bw^4 + b^4w + 2b^2w^3 + w^5$.

Two necklaces are considered identical if there exist symmetries such that the necklaces look the same after the symmetries are applied. For example, the figure below shows three necklaces, each made of two white beads and two gray beads. Necklace 1 and Necklace 2 are distinct since there is no symmetry that can be applied to make them look the same. Necklace 2 and Necklace 3, however, are not considered distinct since a clockwise rotation of 90° applied to Necklace 2 would have it appear identical to Necklace 3.

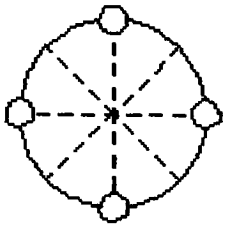


The Dihedral Group

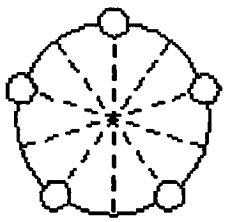
Before discussion of necklace counting can begin, one must first understand the dihedral group. This is the set of rigid motions on an n-gon. It consists of rotations and reflections. The following paragraphs explain dihedral groups in terms of necklaces with an even or odd number of beads.

When the necklace has an even number of beads, it can be rotated a multiple of $(360/n)°$ where n is the number of beads. This gives n distinct rotations. In addition, there are a number of reflections that leave the relative location of beads unchanged. As there are $n/2$ pairs of

opposite beads, we can consider a line of reflection running through these pairs. Finally, there are $n/2$ pairs of opposite edges and we can consider a line of reflection running through these pairs. In total, we see that there are a total of $2n$ symmetries on a necklace of even beads. The following figure shows the lines of reflection for a four-bead necklace.



In the case of an odd number of beads, the number of rotations is similar to that in even necklaces. The lines of reflection, however are not the same. There are no pairs of opposite beads or opposite edges. Instead, a line of reflection is constructed from each bead to the opposite edge. Again, this gives n rotations and n reflections for a total of $2n$ symmetries. The following figure shows the lines of reflection for a five-bead necklace.



Burnside's Theorem

There are two versions of Burnside's Theorem (also known as Not Burnside's Theorem since the results were known to Cauchy and Frobenius decades before Burnside's proof) when applied to the necklace problem. They are as follows:

(1) $N=\frac{1}{|G|}\sum_{x\epsilon T}\phi(x)$

(2) $N=\frac{1}{|G|}\sum_{\pi\epsilon G}\psi(\pi)$

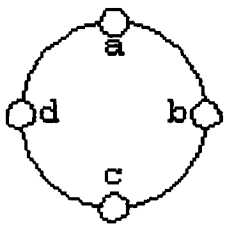
In these equations, N is the number of distinct coloring of a necklace with a given number of colors. $|G|$ is the number of permutation of the necklace. This is the size of the dihedral group, $2n$. Equation (1) makes use of x, T, and ϕ. T is the collection of all coloring of the necklace and

x is any one coloring. $\phi(x)$ denotes the number of symmetries that leave the coloring x fixed. Equation (2) uses G, π, and ψ. G is the group of permutations or symmetries of the necklace and π is any one permutation of symmetry. $\psi(\pi)$ is the number of colorings in T left fixed by π.

What Burnside's Theorem says is that the number of distinct necklaces is equal to the number of colorings left fixed by all symmetries divided by $2n$.

The Cycle Index

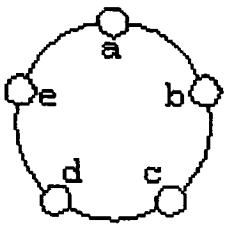
The first step in finding the pattern inventory for a given number of colors and beads is to find the cycle index for a given number of beads. To do so, we must find the cycle structure representation of each symmetry. We can do this for the rotations in a direct manner. For each rotation, we trace the path of the beads through the rotation. Consider the following four-bead necklace.



Consider all rotations to be in a clockwise direction. For a rotation of 0° (we can also consider this a rotation of 360°), each bead remains fixed. Thus, we have a cycle structure of $(a)(b)(c)(d)$. For a rotation of 90°, a goes to b, b goes to c, c goes to d, and d goes to a. This is a cycle structure of $(abcd)$. For rotations of 180° and 270°, we get $(ac)(bd)$ and $(adcb)$. For reflections about the lines through opposite edges, we get $(ab)(cd)$ and $(ad)(bc)$. Finally, for reflections about the lines from a to c and from b to d, we get $(a)(bd)(c)$ and $(ac)(b)(d)$.

The next step is to convert these cycles into a notation in terms of x. Each x will have a subscript and a superscript. x_j^i will appear for each of the i times a cycle of length j occurs in the cycle structure for each symmetry. For the rotations, we get x_1^4, x_4^1, x_2^2, and x_4^1. For the reflections, we get x_2^2, x_2^2, $x_1^2x_2^1$, and $x_1^2x_2^1$. Grouping the terms together gives $x_1^4 + 2x_4^1 + 3x_2^2 + 2x_1^2x_2^1$.

Cycle Index Term	Cycle Structure
x_1^4	$(a)(b)(c)(d)$
$2x_4^1$	$(abcd),(adcb)$
$3x_2^2$	$(ac)(bd),(ab)(cd),(ad)(bc)$
$2x_1^2x_2^1$	$(a)(bd)(c),(ac)(b)(d)$



Now consider a necklace with five beads. Again, there are five clockwise rotations, each a multiple of 72°. For a rotation of 0° (or 360°), we get a cycle structure of $(a)(b)(c)(d)(e)$. For a rotation of 72°, we have a cycle structure of $(abcde)$. In fact, for each rotation other than 0°, we have the cycle structure of $(abcde)$. The reflections about the lines from vertices to opposite edges give the cycle structures $(a)(be)(cd)$, $(b)(ac)(de)$, $(c)(bd)(ae)$, $(d)(ce)(ab)$, and $(e)(ad)(bc)$.

Converting these cycle structures into cycle index notation, we get x_1^5 for 0° and x_5^1 for all other rotations. Each of the reflections gives $x_1^1x_2^2$. Therefore the cycle index is $x_1^5 + 4x_5^1 + 5x_1^1x_2^2$.

Cycle Index Term	**Cycle Structure**
x_1^5	$(a)(b)(c)(d)(e)$
$4x_5^1$	$(abcde),(acebd),(adbec),(aedcb)$
$5x_1^1x_2^2$	$(a)(be)(cd),(b)(ac)(de),(c)(bd)(ae),(d)(ce)(ab),(e)(ad)(bc)$

We can draw some generalizations about computing the cycle index without having to trace the locations through all rotations and reflections. When considering the rotations of necklace with n beads, index them as 1 through n. The cycle index term for the ith rotation will be $x_{n/\gcd(i,n)}^{\gcd(i,n)}$. The cycle index terms for the reflections will depend on whether n is even or odd. For odd n, all the reflection terms will be $x_1^1x_2^{(n-1)/2}$. When n is even, there will be $\frac{n}{2}$ terms of the form $x_2^{n/2}$ and $\frac{n}{2}$ terms of the form $x_1^2x_2^{(n-2)/2}$.

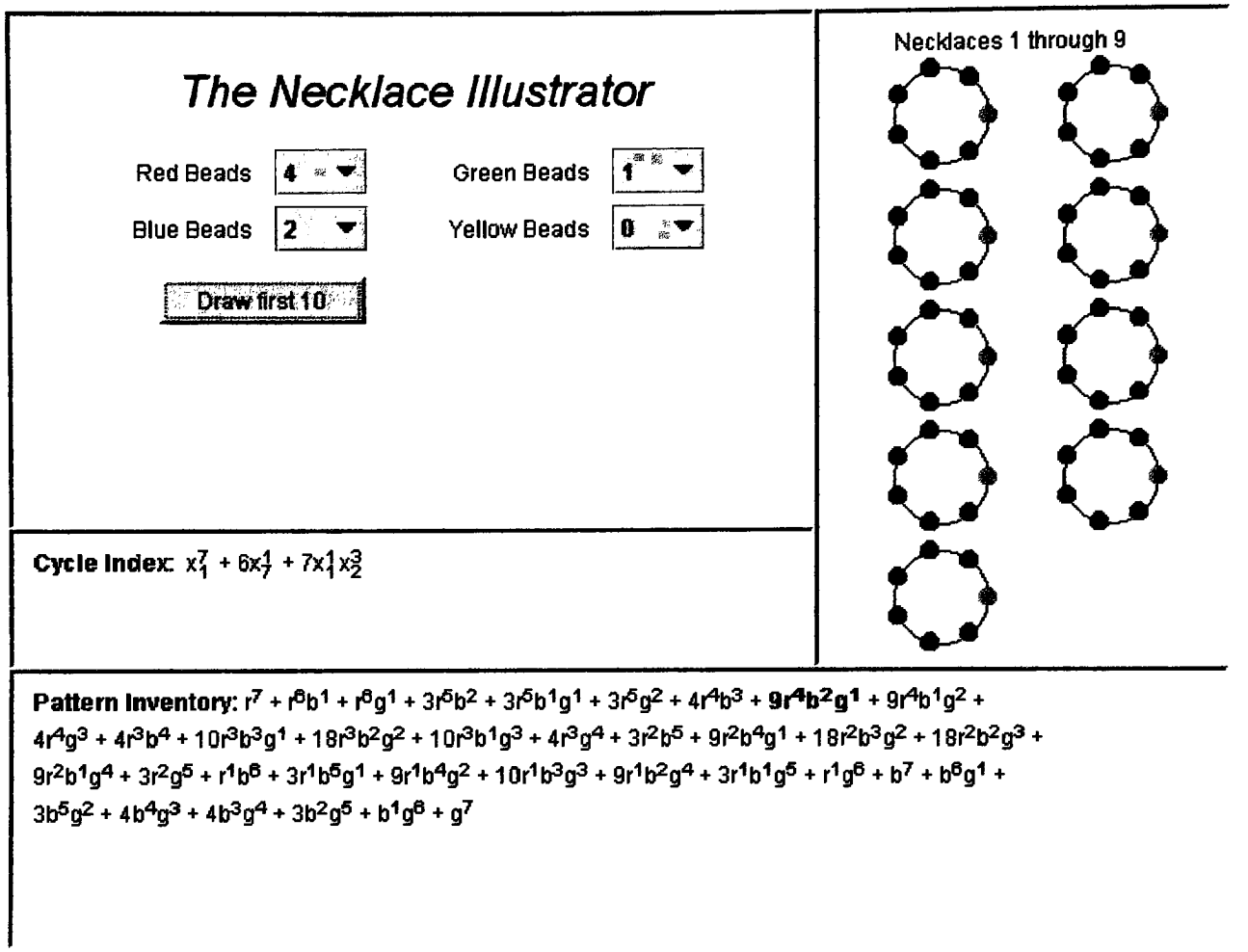
The Pattern Inventory

Once we have computed the cycle index, we are ready to determine the pattern inventory. Construct an expression for each term in the cycle index as follows: $\left(a^d + b^d + \ldots + c^d\right)^e$ such that a, b, ..., c are variables representing different colors, d is the subscript of the cycle index term, and e is the superscript of the cycle index term. Expanding this polynomial will almost result in the pattern inventory for the chosen colors and number of beads. Each distinct necklace appears once for each member of the dihedral group. Therefore, each coefficient will

be too large by a factor of $2n$. Simply dividing the coefficients by $2n$ will give the desired result.

Suppose that we were to color a four-bead necklace with black and white beads. We start with the cycle index for four beads, $x_1^4 + 2x_4^1 + 3x_2^2 + 2x_1^2x_2^1$. Next, we insert $(b^i + w^i)^j$ for each x_i^j. This gives us $(b + w)^4 + 2(b^4 + w^4) + 3(b^2 + w^2)^2 + 2(b + w)^2(b^2 + w^2)$. Following expansion and coefficient division by 8, we are left with an expression of $b^4 + b^3w + 2b^2w^2 + bw^3 + w^4$. Reading the coefficients, this tells us that there is only one distinct necklace with all black beads, one with three black and one white bead, two with two black and two white, one with one black and three white, and one with all white beads.

Section II: The Applet



Introduction to Java

This section contains terms that may not be understood by a reader that does not know much about the Java programming language. This is a brief introduction that explains some of the more important terms used.

Java is an object-oriented language. This means that a program consists of interacting objects, each with certain data and functions predefined. These data and function definitions are determined by the class to which an object belongs. When a new object is created in the program, it is an instance of a class.

A class contains methods. These are the functions that belong to this class. They can be

passed or return data as well as perform operations.

An Applet is a type of Java program. It requires a Java environment, such as a Java- enabled web browser, to run.

When we say that Class A extends Class B, it means that a new class, Class A, was created that inherits all the properties of Class B in addition to any new methods added.

There were some standard Java classes that were used in creating this Applet. One such class is the Vector class. This is essentially an array that contains objects. The difference between the Vector and the usual Java array is that the array must be passed its maximum size upon creation and a Vector is resizable.

Another class repeatedly used was the Canvas class. The Canvas is typically used for drawing to the screen.

Applet Overview

The Applet is designed to display three different results based on the selection of beads chosen by the user. The first result is the cycle index of the total number of beads chosen. The second result is the pattern inventory for the number and colors of beads chosen. Finally, all distinct necklaces based on the selected beads are displayed in groups of up to ten.

Upon loading, the Applet gives the user four drop-down selection boxes from which they can choose the number of beads of each color. The valid colors are red, blue, green, and yellow. The valid choices for each color range from zero to five.

When the user triggers any of the drop-down boxes and selects a number for that color, the Applet automatically computes and displays the new cycle index and pattern inventory. The list of distinct necklaces are not automatically drawn, however. The user must first click a button labelled "Draw first 10". If there are ten or fewer necklaces, they are all drawn on the screen. If there are more than ten necklaces, the "Draw first 10" button is re-labelled "Draw next 10". A new button is also created labelled "New necklace". This second button is included in case the user no longer wishes to view the current necklace and make a new color selection. After a new selection of colors, clicking the "New necklace" button will terminate illustrating the first necklace and will begin drawing the distinct necklaces for the new color selection. When all of the necklaces have been shown, the "Draw next 10" button is once again labelled "Draw first 10" and the "New necklace" button disappears.

Input

The Necklace Applet itself is visually divided into four separate panels. The first is used for input. It contains four JComboBox objects, one for each color. A JComboBox is a "drop-down" menu for selecting data. Each JComboBox has an ActionListener attached to it so that when a new number is selected from that box, the routines for computing cycle index and pattern inventory are automatically run. The input panel contains the "Draw first 10"/"Draw next 10" button and the "New necklace" button. These buttons also have ActionListeners attached to run the appropriate routines.

The panels for the cycle index, pattern inventory, and necklace illustrations contain objects that extend the Canvas class containing modified paint methods.

In the event of an ActionListener being triggered by color selection or a button click, the first thing that occurs that the values in the four JComboBoxes are stored as integer variables for the four colors. The Applet then determines whether it was a color selection or a button click that occurred.

The Cycle Index

When a new color selection is made, a CycleIndex object is created for the total number of beads. This CycleIndex object has the task of creating a Vector that stores CycleIndexList objects. The CycleIndexList is an object that contains an integer coefficient as well as a Vector containing either one or two CycleIndexObj objects. These CycleIndexObj objects represent the '*x*' in the cycle index and store the superscript and subscript. The CycleIndex object has hardcoded routines for total bead values one through twenty.

Once the CycleIndex object is created, the Vector that stores the appropriate cycle index is passed to a CIDraw object. The CIDraw object is a canvas on which the cycle index is drawn to be displayed to the user.

The Pattern Inventory

Next, a PatternInventory object is created and is passed the Vector containing the cycle index as well as the number of red, blue, green, and yellow beads. This object first determines the number of beads used. Specific methods are called for one or two beads. For a total bead count greater than two, specific methods are called based on the number of colors used. All terms of

the pattern inventory are stored in PatternInventoryObj objects. These simply store integer coefficients as well as exponents for the variables *r*, *b*, *g*, and *y*. All PatternInventoryObj terms are stored in a Vector within the PatternInventory object.

If there is one bead, a PatternInventoryObj is created that sets the exponent of the appropriate variable to 1 and is added to the Vector.

If there are two beads, the PatternInventory determines if they are both the same color. If they are, a PatternInventoryObj is created that sets the exponent of the appropriate variable to 2 and is added to the Vector. When there are two beads of two different colors, a PatternInventoryObj is created that sets the exponents of the appropriate variables to 1 and is added to the Vector.

When there are more than two beads, there are four different methods to determine the pattern inventory.

In the simplest case where only one color is used, a PatternInventoryObj object is created that sets the exponent of the appropriate variable to the number of beads and is added to the Vector.

When there are two colors, each CycleIndexList object is read except for the last one, which contains two CycleIndexObj objects (or '*x*' values). For each of these, the superscript is read and used to compute the terms of the polynomial in the following manner: Let *i*=superscript of the cycle index term, *j*=exponent of the first color, *k*=exponent of the second color, and *m*=subscript of the cycle index term. *j* runs from 0 to *i* and *k* runs from *i* to 0 such that $j + k = i$ in each term. A binomial coefficient is produced for each pair of *j* and *k* and all exponents are multiplied by *m*. When this is done for all but the last CycleIndexList, the terms are all grouped together. For the last CycleIndexList, a similar process occurs for each of the two '*x*' terms and they are multiplied together by taking all pairs of terms, adding their exponents, and multiplying their coefficients. All like terms are grouped together for the entire pattern inventory. Finally, all coefficients are divided by $2n$ for the final pattern inventory expression. Each term is stored in a Vector as a PatternInventoryObj object that stores the coefficient as well as the exponents for each color variable.

When there are three or four colors, the procedures are similar. First, a four-dimensional array is created using Vectors. This is done by creating three Vectors that contain Vector objects. The fourth Vector contains PatternInventoryObj objects that store the coefficients and exponents for each color variables. When initially created, the indices of the four Vectors represent the exponents stored in the PatternInventoryObj. The coefficients are all initialized to 0. The CycleIndex is again read for all entries but the last. The terms are expanded in a manner

similar to the two-color case. All exponents for the variables are found such that they add up to the superscript of the CycleIndexObj, a multinomial coefficient is generated, and the exponents are multiplied by the CycleIndexObj subscript. The four-dimensional array is then accessed based on the variable exponents and the current coefficient is added to the coefficient stored in the array. For the last CycleIndexList, each of the '*x*' terms is expanded in a similar manner. For each term in the second '*x*', the terms of the first '*x*' are multiplied by adding the exponents and multiplying the coefficients. The array is then accessed and the current coefficient is added to the coefficient stored in the array. Finally, all PatternInventoryObj objects stored in the array are examined and if the coefficient is not 0, it is written to a Vector that can quickly give all terms of the pattern inventory.

Once the PatternInventory object is created, the Vector that stores the appropriate pattern inventory is passed to a PIDraw object along with the number of beads of each color. The PIDraw object is a Canvas-extending object on which the pattern inventory is drawn to be displayed to the user. When the exponents of the colors match the color selection given by the user, the term is displayed in a bold font.

Necklace Illustrations

When the user selects either of the draw buttons, the Applet creates a new NecklaceCompute object which does the calculations for determining distinct necklaces. Due to restrictions on the larger cases, is must first be determined if the color selection given is small enough to compute. With the cutoff determined to be greater than 14 beads using all four colors, a flag is set. If the selection is too large, a message is displayed saying so. Otherwise the process continues. First, the color selection is transformed into a string representation. For example, 'rrbbbg' would represent two red beads, three blue beads, and one green bead. A partial list of permutations of this string is then generated. This partial list contains strings that represent all distinct necklaces. To reduce the number of permutations, only those that begin with the letter of the smallest color chosen are generated. When all of these are generated, they are examined to see if they also end with the smallest color chosen. If so, they are discounted since they represent necklaces that have undergone a rotation and are a duplicate of some necklace that does not end with the smallest color chosen. With the list of permutations minimized in this manner, each remaining string is passed to a new NecklaceObj object that stores the string, the string reversed, and two integers called the forNum and backNum that represent all consecutive pairs of letters in the forward and backward string. The forNum and backNum are both initialized to 1. Each

forward string is traversed and consecutive letters are considered. Based on what these two letters are, the forNum is multiplied by a certain prime number. Likewise, then backNum is generated by traversing the backward string, considering consecutive letters, and multiplying by a certain prime number. This way, we know what letters are adjacent forward and back, which helps speed up determining duplicate necklaces. These NecklaceObj objects are stored in a Vector.

With the NecklaceObj Vector generated, the Applet can now start eliminating duplicates. To speed up the process, only up to ten distinct necklaces are identified at a time. The NecklaceCompute is passed the Vector of NecklaceObj objects as well as the start and end indices of the desired necklaces. For example, it can be passed 0 to 9, 10 to 19, or 20 to 29. The procedure is done by starting at the lower index and comparing to each prior NecklaceObj, all of which are assumed to be distinct. The forNums are first compared and if they match, there is the possibility that they are duplicate necklaces. The forward string is then rotated by repeatedly moving the leading character to the end. If the strings match at any point, the process is terminated and the current NecklaceObj is marked as a duplicate. If there is no match, the backNum of the current NecklaceObj is compared to the forNum of the prior NecklaceObj. If they are the same, a similar rotation process occurs to look for a match. If it is determined that the current NecklaceObj is a duplicate, it is removed and the next NecklaceObj goes through the same process. If it is determined that the current NecklaceObj is not a duplicate, it is grouped with the prior distinct necklaces and the next NecklaceObj goes through the same process. This repeats until either the upper index is reached or all necklaces are compared.

It is then time to draw the current set of necklaces. A DrawObj object is created that renders the current set of ten or less necklaces. This is done by passing the DrawObj the NecklaceObj Vector and selecting the forward strings for the current set of necklaces. When the first ten necklaces are drawn, a "New necklace" button is created in case the user wishes to view necklaces of a new color selection. Once the entire set of distinct necklaces is drawn, the "New necklace" button is removed.

Section III: Problems and fixes

During the writing of this Applet, a number of problems arose. The following paragraphs document these problems and describe how they were either fixed or worked around.

The first problem encountered was how to generate a list of permutations of a string. This in itself would have been no problem if all the characters were distinct, but the fact that there could be repeated characters meant there would be many repeated permutations. After several hours searching Java books and the Internet, little progress had been made. The key was terminology. When the search was changed from 'permutation' to 'anagram', a solution was soon found. An anagram-generating algorithm was found in *An Introduction to Object-Oriented Programming with Java* by C. Thomas Wu. This algorithm also assumed non-repeated character, but it was easily adapted to take repeated characters into account. This can be found in the NecklaceCompute class in the permutationGenerator() method.

Once this list of permutations was obtained, it was time to start making it more efficient. The list could grow quite large and contain many duplicate necklaces. The first task was to find a way to eliminate as many strings as possible without the risk of losing any distinct necklaces. This was done by considering only strings that start with the letter corresponding to the color with the smallest number of beads. Any other string could be considered a duplicate of one of these (either forward or backward) after some rotation was applied. Therefore, any other string's necklace would be a duplicate of at least one string's necklace from the smaller set. A second check was set up to eliminate further duplicates from this smaller set. The last character of the string is checked and if it also is the letter corresponding to the color with the smallest number of beads, it is discounted. This is due to the fact that the necklace it represents would be a rotation of some necklace whose string did not end in the same letter.

The first problem that was pure Java programming and not algorithm-oriented was that when the distinct necklaces were drawn on the screen and another window was opened over these necklaces, they would disappear when returning to the Applet. This problem existed for several weeks and any attempts to fix it failed. The vital piece of information concerned the Canvas class that was used in drawing these necklaces. When a Canvas-extending object is covered on the screen, it recalls its paint() method upon being exposed again. When nothing is in the paint() method, the default is to clear the screen in that area. The DrawObj class was written extending the Canvas class that contains a custom paint() method which draws the necklaces to the screen. Now each time that the DrawObj object is exposed, the necklaces are redrawn.

When the user selected a necklace composed of just one color, no necklaces would be drawn. This was a result of the method used for obtaining the necklace strings. When a string began and ended with the same character, it was discounted. With only one color used, there is only one string and all characters, including the first and last, are the same. This string would never be considered and an empty Vector would be created. Special code was inserted in the permutation generator of the NecklaceCompute object to handle the case where there was just one bead and the case where there were multiple beads of only one color.

During testing, several combinations of beads were shown to have problems with the number of necklaces display. One of the first to be problematic was a 3-3-2 combination when the user selected two red, three blue, and three green. It displayed a different number of necklaces than when the user selected three red, three blue, and two green, which gave the correct number. The problem was traced back to the omission of an equal sign. When determining whether necklaces were distinct, the NecklaceCompute would go up, but not including, the upper index sent by the Applet. This meant that a few necklaces were not checked and were reported as distinct when they were duplicates.

A problem was found when the Applet was run on some, but not all, Macintosh computers. Problems loading the Applet were found using both the Safari and Internet Explorer browsers. Similar problems were later encountered when attempting to load the Applet onto a Windows computer with Internet Explorer. In the case of the Windows computer, trip to www.sun.com resulted in downloading an up-to-date version of the Java Runtime Environment. After installation, the Applet loaded and performed properly.

A problem was found when trying to display necklaces with a large number of beads. The case that first found this problem was five red, three blue, five green, and four yellow. The problem was tracked back to a Java memory problem. Even when the permutation list is minimized as described above, there are problems with large cases. Although the program discounts many strings as being duplicates, the recursive nature of the permutation generator meant that all permutations were generated, if not saved. A new method was added to the permutation generator that would only build strings that started with the letter of the smallest color. Even with this implemented, there were still problems with the larger cases. The only workaround was to impose a limit so that for these larger cases, the work would not be done that resulted in the memory errors. After testing, the breaking point appeared to be when the user selected more than fourteen beads using all four colors.

Displaying the cycle index and pattern inventory requires the use of subscripts and superscripts. The normal String class output does not allow for their use. The first solution was

to use a StyledDocument and add each piece individually. Each time the style of the text changed, the StyledDocument style had to be changed. This worked for the cycle index, but problems would later arise with the pattern inventory.

The next problem that presented itself was the question of how to expand the pattern inventory polynomial based on the cycle index and colors chosen. The initial attempt mirrored the way in which one would expand polynomials by hand. The terms inside the parenthesis were determined by the colors chosen and the cycle index subscript (for example, $r^4 + b^4 + g^4$). They were then multiplied by itself by taking individual terms and adding the exponents. This was repeated according to the superscript of the cycle index. When this process was completed, all terms were added to a Vector. The Vector would then be traversed and the terms would be grouped together, resulting in a coefficient for each term. While this worked for small cases, it was highly inefficient and failed completely due to memory issues with even moderately large cases. A search was conducted to find if other Java classes could handle taking the expression as a string, expanding the polynomial, and returning the expression as a string that could be displayed. Some were found, but they all proved to be quite complex and learning to use them would be fairly difficult in a short amount of time. Returning to writing a new procedure, the result was the process described in the previous section that can be found in the PatternInventory class.

Once the pattern inventory could be computed and stored in a Vector of PatternInventoryObj objects, a new problem arose concerning how to display them. When the number of beads grows large enough, the panel used to display the terms had problems since the terms were all displayed horizontally by the StyledDocument. In addition, cases that were small enough to display but still fairly large were slow to display. The solution was to render them as a drawing as done with the necklaces. Instead of a StyledDocument, a Canvas-extending object was used that drew the terms on the screen in the panel. The main problem was keeping track of the location where items should be drawn, since the changing styles from regular text to superscript text caused differences in text width measurements.

This approach to writing the pattern inventory worked, but it was still too slow. Each time the screen was refreshed, it would have to write each term individually, which was a problem when there were many terms. For each term, the coefficient was drawn in addition to any variable with a non-zero exponent as well as the exponent. To fix this, a method was included that created a BufferedImage. The terms were then drawn onto this image. Now when the screen is refreshed, it simply redraws one image instead of all the individual terms.

Another round of testing found a problem with the selection of three red, three blue, and one

green. The pattern inventory says that there should be ten distinct necklaces with this color selection. In practice, not only were eight display, but they were labelled "Necklaces 11 through 18". Reviewing the code for determining distinct necklaces found consecutive if statements that had opposite conditions. While the second condition was initially false, executing the code for the first if statement caused it to be true. Changing this to an if-else statement resolved the problem.

The final few problem fixes dealt with the necklace drawing of miscellaneous cases. One such case was when the number of necklaces was a multiple of ten. In some instances, there were additional strings to check for duplicity, so the Applet did not know that all distinct necklaces had already been displayed. If there were only ten, for instance, the final necklace illustration would be blank except for a title that read "Necklaces 11 to 10". This was fixed by adding some code to the Necklace and PIDraw classes. As the PIDraw steps through the pattern inventory terms, it checks to see if the term corresponds to the color selection given by the user. If so, it is drawn in a bold font. In addition, the coefficient is noted as the maximum number of distinct necklaces. The Necklace class uses this coefficient as a cutoff for drawing necklaces.

If the user attempted to draw necklaces with no beads, the draw panel would be blank with the title "Necklaces 1 to 0". Code was inserted in the DrawObj class to instead display the title "Nothing selected".

The final problem encountered was one which was thought of earlier, but dismissed as unlikely to occur. When testing the pattern inventory for twenty beads, five of each color, many of the largest coefficients came up as negative numbers. In Java, the *int* data type has a range up to 2147483647. All of the coefficients were within this range. The problem was that this number must first be divided by 40. Prior to this division, the numbers were outside the range. When this happens, the integer "wraps around" and becomes a negative number. The numbers were changed from the *int* data type to the *long* data type, which has a range up to 9223372036854775807. All numbers were well within this range and displayed properly.

Section IV: The Code

The following pages contain the Java code for all classes used by the Necklace Illustrator Applet.

```
/*
 * CIDraw.java
 * For use with the Necklace Illustrator Applet
 * by David Fraser
 * Rochester Institute of Technology
 * May 2005
 *
 */

package Necklace;

import java.awt.Canvas;
import java.awt.Graphics;
import java.awt.Font;
import java.lang.Integer;
import java.util.Vector;
import Necklace.CycleIndexList;
import Necklace.CycleIndexObj;
import java.awt.FontMetrics;

public class CIDraw extends Canvas{

   Vector ciVector;
   Graphics g;

   //create a new CIDraw that is passed the Cycle Index Vector
   public CIDraw(Vector vec) {
      ciVector = vec;
      g = getGraphics();
   }

   //paint method used to render to screen
   public void paint(Graphics g){

      Font boldFont = new Font("boldFont", java.awt.Font.BOLD, 12);
      g.setFont(boldFont);
      g.drawString("Cycle Index:  ",  10, 20);
      int newX = 10+this.getFontMetrics(boldFont).stringWidth("Cycle Index:  ");
      //if the Cycle Index Vector is not empty
      if (ciVector.size()!=0){
         Font regFont = new Font("regFont", java.awt.Font.PLAIN, 12);
         Font smallFont = new Font("smallFont", java.awt.Font.PLAIN, 10);
         g.setFont(regFont);
         //for each term in the Cycle Index
         for (int i=0; i<ciVector.size(); i++){
```

```
            CycleIndexList ciList = (CycleIndexList)ciVector.elementAt(i);
            Vector tempVector = ciList.getCycleIndexEntry();
            Integer convInt = new Integer(0);
            //if the coefficient is not 1, draw to screen
            if (ciList.getCoeff()!=1){
               String tempStr = convInt.toString(ciList.getCoeff());
               g.drawString(tempStr, newX, 20);
               newX = newX+this.getFontMetrics(regFont).stringWidth(tempStr);
            }
            //for each "x" in term:
            for (int j=0; j<tempVector.size(); j++){
               CycleIndexObj ciObj = (CycleIndexObj)tempVector.elementAt(j);
               g.setFont(regFont);
               //draw "x"
               g.drawString("x", newX, 20);
               newX = 1+newX+this.getFontMetrics(regFont).stringWidth("x");
               g.setFont(smallFont);
               //draw superscript and subscript
               String tempSup = convInt.toString(ciObj.getSup());
               String tempSub = convInt.toString(ciObj.getSub());
               g.drawString(tempSup, newX, 16);
               g.drawString(tempSub, newX, 24);
               newX =newX+ java.lang.Math.max(this.getFontMetrics(smallFont)
               .stringWidth(tempSup),this.getFontMetrics(boldFont)
               .stringWidth(tempSub));
            }
            //if not the last term
            if (i!=ciVector.size()-1){
               g.setFont(regFont);
               //draw " + "
               g.drawString(" + ", newX, 20);
               newX = newX+this.getFontMetrics(regFont).stringWidth(" + ");
            }
         }
      }
      g.dispose();
   }

}
```

```
/*
 * CycleIndex.java
 * For use with the Necklace Illustrator Applet
 * by David Fraser
 * Rochester Institute of Technology
 * May 2005
 *
 */

package Necklace;

import java.util.Vector;

public class CycleIndex {

   private Vector cycleIndexVector;

   //create a new CycleIndex
   public CycleIndex() {
   }

   //create a new CycleIndex that is passed the number of beads
   public CycleIndex(int beads){
      cycleIndexVector = new Vector();
      //check the number of beads, call appropriate method
      switch (beads)  {
         case 1:  oneBead();
                  break;
         case 2:  twoBead();
                  break;
         case 3:  threeBead();
                  break;
         case 4:  fourBead();
                  break;
         case 5:  fiveBead();
                  break;
         case 6:  sixBead();
                  break;
         case 7:  sevenBead();
                  break;
         case 8:  eightBead();
                  break;
         case 9:  nineBead();
                  break;
         case 10: tenBead();
```

```
            break;
        case 11: elevenBead();
            break;
        case 12: twelveBead();
            break;
        case 13: thirteenBead();
            break;
        case 14: fourteenBead();
            break;
        case 15: fifteenBead();
            break;
        case 16: sixteenBead();
            break;
        case 17: seventeenBead();
            break;
        case 18: eighteenBead();
            break;
        case 19: nineteenBead();
            break;
        case 20: twentyBead();
            break;
    }
}

private void oneBead(){
    //create x(sub(1), sup(1))
    CycleIndexList tempList = new CycleIndexList();
    CycleIndexObj tempObj = new CycleIndexObj(1,1);
    tempList.addCI(tempObj);
    tempList.setCoeff(1);
    cycleIndexVector.add(tempList);
}

private void twoBead(){
    //create x(sub(1), sup(2))
    CycleIndexList tempListOne = new CycleIndexList();
    CycleIndexObj tempObjOne = new CycleIndexObj(2,1);
    tempListOne.addCI(tempObjOne);
    tempListOne.setCoeff(1);
    cycleIndexVector.add(tempListOne);

    //create x(sub(2), sup(1))
    CycleIndexList tempListTwo = new CycleIndexList();
    CycleIndexObj tempObjTwo = new CycleIndexObj(1,2);
    tempListTwo.addCI(tempObjTwo);
```

```
    tempListTwo.setCoeff(1);
    cycleIndexVector.add(tempListTwo);
}

private void threeBead(){
    //create x(sub(1), sup(3))
    CycleIndexList tempListOne = new CycleIndexList();
    CycleIndexObj tempObjOne = new CycleIndexObj(3,1);
    tempListOne.addCI(tempObjOne);
    tempListOne.setCoeff(1);
    cycleIndexVector.add(tempListOne);

    //create 2*x(sub(3), sup(1))
    CycleIndexList tempListTwo = new CycleIndexList();
    CycleIndexObj tempObjTwo = new CycleIndexObj(1,3);
    tempListTwo.addCI(tempObjTwo);
    tempListTwo.setCoeff(2);
    cycleIndexVector.add(tempListTwo);

    //create 3*x(sub(1), sup(1))*x(sub(2), sup(1))
    CycleIndexList tempListThree = new CycleIndexList();
    CycleIndexObj tempObjThreeA = new CycleIndexObj(1,1);
    CycleIndexObj tempObjThreeB = new CycleIndexObj(1,2);
    tempListThree.addCI(tempObjThreeA);
    tempListThree.addCI(tempObjThreeB);
    tempListThree.setCoeff(3);
    cycleIndexVector.add(tempListThree);
}

private void fourBead(){
    //create x(sub(1), sup(4))
    CycleIndexList tempListOne = new CycleIndexList();
    CycleIndexObj tempObjOne = new CycleIndexObj(4,1);
    tempListOne.addCI(tempObjOne);
    tempListOne.setCoeff(1);
    cycleIndexVector.add(tempListOne);

    //create 2*x(sub(4), sup(1))
    CycleIndexList tempListTwo = new CycleIndexList();
    CycleIndexObj tempObjTwo = new CycleIndexObj(1,4);
    tempListTwo.addCI(tempObjTwo);
    tempListTwo.setCoeff(2);
    cycleIndexVector.add(tempListTwo);
```

```
    //create 3*x(sub(2), sup(2))
    CycleIndexList tempListThree = new CycleIndexList();
    CycleIndexObj tempObjThree = new CycleIndexObj(2,2);
    tempListThree.addCI(tempObjThree);
    tempListThree.setCoeff(3);
    cycleIndexVector.add(tempListThree);

    //create 2*x(sub(1), sup(2))*x(sub(2), sup(1))
    CycleIndexList tempListFour = new CycleIndexList();
    CycleIndexObj tempObjFourA = new CycleIndexObj(2,1);
    CycleIndexObj tempObjFourB = new CycleIndexObj(1,2);
    tempListFour.addCI(tempObjFourA);
    tempListFour.addCI(tempObjFourB);
    tempListFour.setCoeff(2);
    cycleIndexVector.add(tempListFour);
  }

  private void fiveBead(){
    //create x(sub(1), sup(5))
    CycleIndexList tempListOne = new CycleIndexList();
    CycleIndexObj tempObjOne = new CycleIndexObj(5,1);
    tempListOne.addCI(tempObjOne);
    tempListOne.setCoeff(1);
    cycleIndexVector.add(tempListOne);

    //create 4*x(sub(5), sup(1))
    CycleIndexList tempListTwo = new CycleIndexList();
    CycleIndexObj tempObjTwo = new CycleIndexObj(1,5);
    tempListTwo.addCI(tempObjTwo);
    tempListTwo.setCoeff(4);
    cycleIndexVector.add(tempListTwo);

    //create 5*x(sub(1), sup(1))*x(sub(2), sup(2))
    CycleIndexList tempListThree = new CycleIndexList();
    CycleIndexObj tempObjThreeA = new CycleIndexObj(1,1);
    CycleIndexObj tempObjThreeB = new CycleIndexObj(2,2);
    tempListThree.addCI(tempObjThreeA);
    tempListThree.addCI(tempObjThreeB);
    tempListThree.setCoeff(5);
    cycleIndexVector.add(tempListThree);
  }

  private void sixBead(){
    //create x(sub(1), sup(6))
    CycleIndexList tempListOne = new CycleIndexList();
```

```
    CycleIndexObj tempObjOne = new CycleIndexObj(6,1);
    tempListOne.addCI(tempObjOne);
    tempListOne.setCoeff(1);
    cycleIndexVector.add(tempListOne);

    //create 2*x(sub(6), sup(1))
    CycleIndexList tempListTwo = new CycleIndexList();
    CycleIndexObj tempObjTwo = new CycleIndexObj(1,6);
    tempListTwo.addCI(tempObjTwo);
    tempListTwo.setCoeff(2);
    cycleIndexVector.add(tempListTwo);

    //create 2*x(sub(3), sub(2))
    CycleIndexList tempListThree = new CycleIndexList();
    CycleIndexObj tempObjThree = new CycleIndexObj(2,3);
    tempListThree.addCI(tempObjThree);
    tempListThree.setCoeff(2);
    cycleIndexVector.add(tempListThree);

    //create 4*x(sub(2), sup(3))
    CycleIndexList tempListFour = new CycleIndexList();
    CycleIndexObj tempObjFour = new CycleIndexObj(3,2);
    tempListFour.addCI(tempObjFour);
    tempListFour.setCoeff(4);
    cycleIndexVector.add(tempListFour);

    //create 3*x(sub(1), sup(2))*x(sub(2), sup(2))
    CycleIndexList tempListFive = new CycleIndexList();
    CycleIndexObj tempObjFiveA = new CycleIndexObj(2,1);
    CycleIndexObj tempObjFiveB = new CycleIndexObj(2,2);
    tempListFive.addCI(tempObjFiveA);
    tempListFive.addCI(tempObjFiveB);
    tempListFive.setCoeff(3);
    cycleIndexVector.add(tempListFive);
}

private void sevenBead(){
    //create x(sub(1), sup(7))
    CycleIndexList tempListOne = new CycleIndexList();
    CycleIndexObj tempObjOne = new CycleIndexObj(7,1);
    tempListOne.addCI(tempObjOne);
    tempListOne.setCoeff(1);
    cycleIndexVector.add(tempListOne);
```

```
    //create 6*x(sub(7), sup(1))
    CycleIndexList tempListTwo = new CycleIndexList();
    CycleIndexObj tempObjTwo = new CycleIndexObj(1,7);
    tempListTwo.addCI(tempObjTwo);
    tempListTwo.setCoeff(6);
    cycleIndexVector.add(tempListTwo);

    //create 7*x(sub(1), sup(1))*x(sub(2), sup(3))
    CycleIndexList tempListThree = new CycleIndexList();
    CycleIndexObj tempObjThreeA = new CycleIndexObj(1,1);
    CycleIndexObj tempObjThreeB = new CycleIndexObj(3,2);
    tempListThree.addCI(tempObjThreeA);
    tempListThree.addCI(tempObjThreeB);
    tempListThree.setCoeff(7);
    cycleIndexVector.add(tempListThree);
}

private void eightBead(){
    //create x(sub(1), sup(8))
    CycleIndexList tempListOne = new CycleIndexList();
    CycleIndexObj tempObjOne = new CycleIndexObj(8,1);
    tempListOne.addCI(tempObjOne);
    tempListOne.setCoeff(1);
    cycleIndexVector.add(tempListOne);

    //create 4*x(sub(8), sup(1))
    CycleIndexList tempListTwo = new CycleIndexList();
    CycleIndexObj tempObjTwo = new CycleIndexObj(1,8);
    tempListTwo.addCI(tempObjTwo);
    tempListTwo.setCoeff(4);
    cycleIndexVector.add(tempListTwo);

    //create 2*x(sub(4), sup(2))
    CycleIndexList tempListThree = new CycleIndexList();
    CycleIndexObj tempObjThree = new CycleIndexObj(2,4);
    tempListThree.addCI(tempObjThree);
    tempListThree.setCoeff(2);
    cycleIndexVector.add(tempListThree);

    //create 5*x(sub(2), sup(4))
    CycleIndexList tempListFour = new CycleIndexList();
    CycleIndexObj tempObjFour = new CycleIndexObj(4,2);
    tempListFour.addCI(tempObjFour);
    tempListFour.setCoeff(5);
    cycleIndexVector.add(tempListFour);
```

```
    //create 4*x(sub(1), sup(2))*x(sub(2), sup(3))
    CycleIndexList tempListFive = new CycleIndexList();
    CycleIndexObj tempObjFiveA = new CycleIndexObj(2,1);
    CycleIndexObj tempObjFiveB = new CycleIndexObj(3,2);
    tempListFive.addCI(tempObjFiveA);
    tempListFive.addCI(tempObjFiveB);
    tempListFive.setCoeff(4);
    cycleIndexVector.add(tempListFive);
}

private void nineBead(){
    //create x(sub(1), sup(9))
    CycleIndexList tempListOne = new CycleIndexList();
    CycleIndexObj tempObjOne = new CycleIndexObj(9,1);
    tempListOne.addCI(tempObjOne);
    tempListOne.setCoeff(1);
    cycleIndexVector.add(tempListOne);

    //create 6*x(sub(9), sup(1))
    CycleIndexList tempListTwo = new CycleIndexList();
    CycleIndexObj tempObjTwo = new CycleIndexObj(1,9);
    tempListTwo.addCI(tempObjTwo);
    tempListTwo.setCoeff(6);
    cycleIndexVector.add(tempListTwo);

    //create 2*x(sub(3), sup(3))
    CycleIndexList tempListThree = new CycleIndexList();
    CycleIndexObj tempObjThree = new CycleIndexObj(3,3);
    tempListThree.addCI(tempObjThree);
    tempListThree.setCoeff(2);
    cycleIndexVector.add(tempListThree);

    //create 9*x(sub(1), sup(1))*x(sub(2), sup(4))
    CycleIndexList tempListFour = new CycleIndexList();
    CycleIndexObj tempObjFourA = new CycleIndexObj(1,1);
    CycleIndexObj tempObjFourB = new CycleIndexObj(4,2);
    tempListFour.addCI(tempObjFourA);
    tempListFour.addCI(tempObjFourB);
    tempListFour.setCoeff(9);
    cycleIndexVector.add(tempListFour);
}

private void tenBead(){
    //create x(sub(1), sup(10))
```

```
    CycleIndexList tempListOne = new CycleIndexList();
    CycleIndexObj tempObjOne = new CycleIndexObj(10,1);
    tempListOne.addCI(tempObjOne);
    tempListOne.setCoeff(1);
    cycleIndexVector.add(tempListOne);

    //create 4*x(sub(10), sup(1))
    CycleIndexList tempListTwo = new CycleIndexList();
    CycleIndexObj tempObjTwo = new CycleIndexObj(1,10);
    tempListTwo.addCI(tempObjTwo);
    tempListTwo.setCoeff(4);
    cycleIndexVector.add(tempListTwo);

    //create 4*x(sub(5), sup(2))
    CycleIndexList tempListThree = new CycleIndexList();
    CycleIndexObj tempObjThree = new CycleIndexObj(2,5);
    tempListThree.addCI(tempObjThree);
    tempListThree.setCoeff(4);
    cycleIndexVector.add(tempListThree);

    //create 6*x(sub(2), sup(5))
    CycleIndexList tempListFour = new CycleIndexList();
    CycleIndexObj tempObjFour = new CycleIndexObj(5,2);
    tempListFour.addCI(tempObjFour);
    tempListFour.setCoeff(6);
    cycleIndexVector.add(tempListFour);

    //create 5*x(sub(1), sup(2))*x(sub(2), sup(4))
    CycleIndexList tempListFive = new CycleIndexList();
    CycleIndexObj tempObjFiveA = new CycleIndexObj(2,1);
    CycleIndexObj tempObjFiveB = new CycleIndexObj(4,2);
    tempListFive.addCI(tempObjFiveA);
    tempListFive.addCI(tempObjFiveB);
    tempListFive.setCoeff(5);
    cycleIndexVector.add(tempListFive);
}

private void elevenBead(){
    //create x(sub(1), sup(11))
    CycleIndexList tempListOne = new CycleIndexList();
    CycleIndexObj tempObjOne = new CycleIndexObj(11,1);
    tempListOne.addCI(tempObjOne);
    tempListOne.setCoeff(1);
    cycleIndexVector.add(tempListOne);
```

```
    //create 10*x(sub(11), sup(1))
    CycleIndexList tempListTwo = new CycleIndexList();
    CycleIndexObj tempObjTwo = new CycleIndexObj(1,11);
    tempListTwo.addCI(tempObjTwo);
    tempListTwo.setCoeff(10);
    cycleIndexVector.add(tempListTwo);

    //create 11*x(sub(1), sup(1))*x(sub(2), sup(5))
    CycleIndexList tempListThree = new CycleIndexList();
    CycleIndexObj tempObjThreeA = new CycleIndexObj(1,1);
    CycleIndexObj tempObjThreeB = new CycleIndexObj(5,2);
    tempListThree.addCI(tempObjThreeA);
    tempListThree.addCI(tempObjThreeB);
    tempListThree.setCoeff(11);
    cycleIndexVector.add(tempListThree);
}

private void twelveBead(){
    //create x(sub(1), sup(12))
    CycleIndexList tempListOne = new CycleIndexList();
    CycleIndexObj tempObjOne = new CycleIndexObj(12,1);
    tempListOne.addCI(tempObjOne);
    tempListOne.setCoeff(1);
    cycleIndexVector.add(tempListOne);

    //create 4*x(sub(12), sup(1))
    CycleIndexList tempListTwo = new CycleIndexList();
    CycleIndexObj tempObjTwo = new CycleIndexObj(1,12);
    tempListTwo.addCI(tempObjTwo);
    tempListTwo.setCoeff(4);
    cycleIndexVector.add(tempListTwo);

    //create 2*x(sub(6), sup(2))
    CycleIndexList tempListThree = new CycleIndexList();
    CycleIndexObj tempObjThree = new CycleIndexObj(2,6);
    tempListThree.addCI(tempObjThree);
    tempListThree.setCoeff(2);
    cycleIndexVector.add(tempListThree);

    //create 2*x(sub(4), sup(3))
    CycleIndexList tempListFour = new CycleIndexList();
    CycleIndexObj tempObjFour = new CycleIndexObj(3,4);
    tempListFour.addCI(tempObjFour);
    tempListFour.setCoeff(2);
    cycleIndexVector.add(tempListFour);
```

```
    //create 2*x(sub(3), sup(4))
    CycleIndexList tempListFive = new CycleIndexList();
    CycleIndexObj tempObjFive = new CycleIndexObj(4,3);
    tempListFive.addCI(tempObjFive);
    tempListFive.setCoeff(2);
    cycleIndexVector.add(tempListFive);

    //create 7*x(sub(2), sup(6))
    CycleIndexList tempListSix = new CycleIndexList();
    CycleIndexObj tempObjSix = new CycleIndexObj(6,2);
    tempListSix.addCI(tempObjSix);
    tempListSix.setCoeff(7);
    cycleIndexVector.add(tempListSix);

    //create 6*x(sub(1), sup(2))*x(sub(2), sup(5))
    CycleIndexList tempListSeven = new CycleIndexList();
    CycleIndexObj tempObjSevenA = new CycleIndexObj(2,1);
    CycleIndexObj tempObjSevenB = new CycleIndexObj(5,2);
    tempListSeven.addCI(tempObjSevenA);
    tempListSeven.addCI(tempObjSevenB);
    tempListSeven.setCoeff(6);
    cycleIndexVector.add(tempListSeven);
}

private void thirteenBead(){
    //create x(sub(1), sup(13))
    CycleIndexList tempListOne = new CycleIndexList();
    CycleIndexObj tempObjOne = new CycleIndexObj(13,1);
    tempListOne.addCI(tempObjOne);
    tempListOne.setCoeff(1);
    cycleIndexVector.add(tempListOne);

    //create 12*x(sub(13), sup(1))
    CycleIndexList tempListTwo = new CycleIndexList();
    CycleIndexObj tempObjTwo = new CycleIndexObj(1,13);
    tempListTwo.addCI(tempObjTwo);
    tempListTwo.setCoeff(12);
    cycleIndexVector.add(tempListTwo);

    //create 13*x(sub(1), sup(1))*x(sub(2), sup(6))
    CycleIndexList tempListThree = new CycleIndexList();
    CycleIndexObj tempObjThreeA = new CycleIndexObj(1,1);
    CycleIndexObj tempObjThreeB = new CycleIndexObj(6,2);
    tempListThree.addCI(tempObjThreeA);
```

```
    tempListThree.addCI(tempObjThreeB);
    tempListThree.setCoeff(13);
    cycleIndexVector.add(tempListThree);
  }

  private void fourteenBead(){
    //create x(sub(1), sup(14))
    CycleIndexList tempListOne = new CycleIndexList();
    CycleIndexObj tempObjOne = new CycleIndexObj(14,1);
    tempListOne.addCI(tempObjOne);
    tempListOne.setCoeff(1);
    cycleIndexVector.add(tempListOne);

    //create 6*x(sub(14), sup(1))
    CycleIndexList tempListTwo = new CycleIndexList();
    CycleIndexObj tempObjTwo = new CycleIndexObj(1,14);
    tempListTwo.addCI(tempObjTwo);
    tempListTwo.setCoeff(6);
    cycleIndexVector.add(tempListTwo);

    //create 6*x(sub(7), sup(2))
    CycleIndexList tempListThree = new CycleIndexList();
    CycleIndexObj tempObjThree = new CycleIndexObj(2,7);
    tempListThree.addCI(tempObjThree);
    tempListThree.setCoeff(6);
    cycleIndexVector.add(tempListThree);

    //create 8*x(sub(2), sup(7))
    CycleIndexList tempListFour = new CycleIndexList();
    CycleIndexObj tempObjFour = new CycleIndexObj(7,2);
    tempListFour.addCI(tempObjFour);
    tempListFour.setCoeff(8);
    cycleIndexVector.add(tempListFour);

    //create 7*x(sub(1), sup(2))*x(sub(2), sup(6))
    CycleIndexList tempListFive = new CycleIndexList();
    CycleIndexObj tempObjFiveA = new CycleIndexObj(2,1);
    CycleIndexObj tempObjFiveB = new CycleIndexObj(6,2);
    tempListFive.addCI(tempObjFiveA);
    tempListFive.addCI(tempObjFiveB);
    tempListFive.setCoeff(7);
    cycleIndexVector.add(tempListFive);
  }

  private void fifteenBead(){
```

```
    //create x(sub(1), sup(15))
    CycleIndexList tempListOne = new CycleIndexList();
    CycleIndexObj tempObjOne = new CycleIndexObj(15,1);
    tempListOne.addCI(tempObjOne);
    tempListOne.setCoeff(1);
    cycleIndexVector.add(tempListOne);

    //create 8*x(sub(15), sup(1))
    CycleIndexList tempListTwo = new CycleIndexList();
    CycleIndexObj tempObjTwo = new CycleIndexObj(1,15);
    tempListTwo.addCI(tempObjTwo);
    tempListTwo.setCoeff(8);
    cycleIndexVector.add(tempListTwo);

    //create 4*x(sub(5), sup(3))
    CycleIndexList tempListThree = new CycleIndexList();
    CycleIndexObj tempObjThree = new CycleIndexObj(3,5);
    tempListThree.addCI(tempObjThree);
    tempListThree.setCoeff(4);
    cycleIndexVector.add(tempListThree);

    //create 2*x(sub(3), sup(5))
    CycleIndexList tempListFour = new CycleIndexList();
    CycleIndexObj tempObjFour = new CycleIndexObj(5,3);
    tempListFour.addCI(tempObjFour);
    tempListFour.setCoeff(2);
    cycleIndexVector.add(tempListFour);

    //create 15*x(sub(1), sup(1))*x(sub(2), sup(7))
    CycleIndexList tempListFive = new CycleIndexList();
    CycleIndexObj tempObjFiveA = new CycleIndexObj(1,1);
    CycleIndexObj tempObjFiveB = new CycleIndexObj(7,2);
    tempListFive.addCI(tempObjFiveA);
    tempListFive.addCI(tempObjFiveB);
    tempListFive.setCoeff(15);
    cycleIndexVector.add(tempListFive);
}

private void sixteenBead(){
    //create x(sub(1), sup(16))
    CycleIndexList tempListOne = new CycleIndexList();
    CycleIndexObj tempObjOne = new CycleIndexObj(16,1);
    tempListOne.addCI(tempObjOne);
    tempListOne.setCoeff(1);
    cycleIndexVector.add(tempListOne);
```

```
    //create 8*x(sub(16), sup(1))
    CycleIndexList tempListTwo = new CycleIndexList();
    CycleIndexObj tempObjTwo = new CycleIndexObj(1,16);
    tempListTwo.addCI(tempObjTwo);
    tempListTwo.setCoeff(8);
    cycleIndexVector.add(tempListTwo);

    //create 4*x(sub(8), sup(2))
    CycleIndexList tempListThree = new CycleIndexList();
    CycleIndexObj tempObjThree = new CycleIndexObj(2,8);
    tempListThree.addCI(tempObjThree);
    tempListThree.setCoeff(4);
    cycleIndexVector.add(tempListThree);

    //create 2*x(sub(4), sup(4))
    CycleIndexList tempListFour = new CycleIndexList();
    CycleIndexObj tempObjFour = new CycleIndexObj(4,4);
    tempListFour.addCI(tempObjFour);
    tempListFour.setCoeff(2);
    cycleIndexVector.add(tempListFour);

    //create 9*x(sub(2), sup(8))
    CycleIndexList tempListFive = new CycleIndexList();
    CycleIndexObj tempObjFive = new CycleIndexObj(8,2);
    tempListFive.addCI(tempObjFive);
    tempListFive.setCoeff(9);
    cycleIndexVector.add(tempListFive);

    //create 8*x(sub(1), sup(2))*x(sub(2), sup(7))
    CycleIndexList tempListSix = new CycleIndexList();
    CycleIndexObj tempObjSixA = new CycleIndexObj(2,1);
    CycleIndexObj tempObjSixB = new CycleIndexObj(7,2);
    tempListSix.addCI(tempObjSixA);
    tempListSix.addCI(tempObjSixB);
    tempListSix.setCoeff(8);
    cycleIndexVector.add(tempListSix);
}

private void seventeenBead(){
    //create x(sub(1), sup(17))
    CycleIndexList tempListOne = new CycleIndexList();
    CycleIndexObj tempObjOne = new CycleIndexObj(17,1);
    tempListOne.addCI(tempObjOne);
    tempListOne.setCoeff(1);
```

```
    cycleIndexVector.add(tempListOne);

    //create 16*x(sub(17), sup(1))
    CycleIndexList tempListTwo = new CycleIndexList();
    CycleIndexObj tempObjTwo = new CycleIndexObj(1,17);
    tempListTwo.addCI(tempObjTwo);
    tempListTwo.setCoeff(16);
    cycleIndexVector.add(tempListTwo);

    //create 17*x(sub(1), sup(1))*x(sub(2), sup(8))
    CycleIndexList tempListThree = new CycleIndexList();
    CycleIndexObj tempObjThreeA = new CycleIndexObj(1,1);
    CycleIndexObj tempObjThreeB = new CycleIndexObj(8,2);
    tempListThree.addCI(tempObjThreeA);
    tempListThree.addCI(tempObjThreeB);
    tempListThree.setCoeff(17);
    cycleIndexVector.add(tempListThree);
}

private void eighteenBead(){
    //create x(sub(1), sup(18))
    CycleIndexList tempListOne = new CycleIndexList();
    CycleIndexObj tempObjOne = new CycleIndexObj(18,1);
    tempListOne.addCI(tempObjOne);
    tempListOne.setCoeff(1);
    cycleIndexVector.add(tempListOne);

    //create 6*x(sub(18), sup(1))
    CycleIndexList tempListTwo = new CycleIndexList();
    CycleIndexObj tempObjTwo = new CycleIndexObj(1,18);
    tempListTwo.addCI(tempObjTwo);
    tempListTwo.setCoeff(6);
    cycleIndexVector.add(tempListTwo);

    //create 6*x(sub(9), sup(2))
    CycleIndexList tempListThree = new CycleIndexList();
    CycleIndexObj tempObjThree = new CycleIndexObj(2,9);
    tempListThree.addCI(tempObjThree);
    tempListThree.setCoeff(6);
    cycleIndexVector.add(tempListThree);

    //create 2*x(sub(6), sup(3))
    CycleIndexList tempListFour = new CycleIndexList();
    CycleIndexObj tempObjFour = new CycleIndexObj(3,6);
    tempListFour.addCI(tempObjFour);
```

```
    tempListFour.setCoeff(2);
    cycleIndexVector.add(tempListFour);

    //create 2*x(sub(3), sup(6))
    CycleIndexList tempListFive = new CycleIndexList();
    CycleIndexObj tempObjFive = new CycleIndexObj(6,3);
    tempListFive.addCI(tempObjFive);
    tempListFive.setCoeff(2);
    cycleIndexVector.add(tempListFive);

    //create 10*x(sub(2), sup(9))
    CycleIndexList tempListSix = new CycleIndexList();
    CycleIndexObj tempObjSix = new CycleIndexObj(9,2);
    tempListSix.addCI(tempObjSix);
    tempListSix.setCoeff(10);
    cycleIndexVector.add(tempListSix);

    //create 9*x(sub(1), sup(2))*x(sub(2), sup(8))
    CycleIndexList tempListSeven = new CycleIndexList();
    CycleIndexObj tempObjSevenA = new CycleIndexObj(2,1);
    CycleIndexObj tempObjSevenB = new CycleIndexObj(8,2);
    tempListSeven.addCI(tempObjSevenA);
    tempListSeven.addCI(tempObjSevenB);
    tempListSeven.setCoeff(9);
    cycleIndexVector.add(tempListSeven);
}

private void nineteenBead(){
    //create x(sub(1), sup(19))
    CycleIndexList tempListOne = new CycleIndexList();
    CycleIndexObj tempObjOne = new CycleIndexObj(19,1);
    tempListOne.addCI(tempObjOne);
    tempListOne.setCoeff(1);
    cycleIndexVector.add(tempListOne);

    //create 18*x(sub(19), sup(1))
    CycleIndexList tempListTwo = new CycleIndexList();
    CycleIndexObj tempObjTwo = new CycleIndexObj(1,19);
    tempListTwo.addCI(tempObjTwo);
    tempListTwo.setCoeff(18);
    cycleIndexVector.add(tempListTwo);

    //create 19*x(sub(1), sup(1))*x(sub(2), sup(9))
    CycleIndexList tempListThree = new CycleIndexList();
    CycleIndexObj tempObjThreeA = new CycleIndexObj(1,1);
```

```
    CycleIndexObj tempObjThreeB = new CycleIndexObj(9,2);
    tempListThree.addCI(tempObjThreeA);
    tempListThree.addCI(tempObjThreeB);
    tempListThree.setCoeff(19);
    cycleIndexVector.add(tempListThree);
}

private void twentyBead(){
    //create x(sub(1), sup(20))
    CycleIndexList tempListOne = new CycleIndexList();
    CycleIndexObj tempObjOne = new CycleIndexObj(20,1);
    tempListOne.addCI(tempObjOne);
    tempListOne.setCoeff(1);
    cycleIndexVector.add(tempListOne);

    //create 8*x(sub(20), sup(1))
    CycleIndexList tempListTwo = new CycleIndexList();
    CycleIndexObj tempObjTwo = new CycleIndexObj(1,20);
    tempListTwo.addCI(tempObjTwo);
    tempListTwo.setCoeff(8);
    cycleIndexVector.add(tempListTwo);

    //create 4*x(sub(10), sup(2))
    CycleIndexList tempListThree = new CycleIndexList();
    CycleIndexObj tempObjThree = new CycleIndexObj(2,10);
    tempListThree.addCI(tempObjThree);
    tempListThree.setCoeff(4);
    cycleIndexVector.add(tempListThree);

    //create 4*x(sub(5), sup(4))
    CycleIndexList tempListFour = new CycleIndexList();
    CycleIndexObj tempObjFour = new CycleIndexObj(4,5);
    tempListFour.addCI(tempObjFour);
    tempListFour.setCoeff(4);
    cycleIndexVector.add(tempListFour);

    //create 2*x(sub(4), sup(5))
    CycleIndexList tempListFive = new CycleIndexList();
    CycleIndexObj tempObjFive = new CycleIndexObj(5,4);
    tempListFive.addCI(tempObjFive);
    tempListFive.setCoeff(2);
    cycleIndexVector.add(tempListFive);

    //create 11*x(sub(2), sup(10))
    CycleIndexList tempListSix = new CycleIndexList();
```

```
        CycleIndexObj tempObjSix = new CycleIndexObj(10,2);
        tempListSix.addCI(tempObjSix);
        tempListSix.setCoeff(11);
        cycleIndexVector.add(tempListSix);

        //create 10*x(sub(1), sup(2))*x(sub(2), sup(9))
        CycleIndexList tempListSeven = new CycleIndexList();
        CycleIndexObj tempObjSevenA = new CycleIndexObj(2,1);
        CycleIndexObj tempObjSevenB = new CycleIndexObj(9,2);
        tempListSeven.addCI(tempObjSevenA);
        tempListSeven.addCI(tempObjSevenB);
        tempListSeven.setCoeff(10);
        cycleIndexVector.add(tempListSeven);
    }

    public Vector getCIVector(){
        //return the Cycle Index Vector
        return cycleIndexVector;
    }

}
```

```
/*
 * CycleIndexList.java
 * For use with the Necklace Illustrator Applet
 * by David Fraser
 * Rochester Institute of Technology
 * May 2005
 *
 */

package Necklace;

import java.util.Vector;

public class CycleIndexList {

   private Vector cycleIndexEntry ;
   private int coeff;

   //create a new CycleIndexList
   public CycleIndexList() {
      //create a new Vector to store Cycle Index "x" terms
      cycleIndexEntry = new Vector();
   }

   public void setCoeff(int i){
      //set the coefficient of the Cycle Index term
      coeff = i;
   }

   public int getCoeff(){
      //return the coefficient of the Cycle Index term
      return coeff;
   }

   public void addCI(CycleIndexObj cio){
      //add an "x" term to the Cycle Index term
      cycleIndexEntry.add(cio);
   }

   public Vector getCycleIndexEntry(){
      //return the "x" terms of the Cycle Index term
      return cycleIndexEntry;
   }

}
```

```
/*
 * CycleIndexObj.java
 * For use with the Necklace Illustrator Applet
 * by David Fraser
 * Rochester Institute of Technology
 * May 2005
 *
 */

package Necklace;

public class CycleIndexObj {

   private int sup;
   private int sub;

   //create a new CycleIndexObj
   public CycleIndexObj(){
   }

   //create a new CycleIndexObj that is passed the superscript and subscript
   public CycleIndexObj(int a, int b) {
      sup = a;
      sub = b;
   }

   public void setSub(int a){
      //set the subcript of the "x" term
      sub = a;
   }

   public void setSup(int a){
      //set the superscript of the "x" term
      sup = a;
   }

   public int getSub(){
      //return the subscript of the "x" term
      return sub;
   }

   public int getSup(){
      //return the superscript of the "x" term
      return sup;
   }}
```

```
/*
 * DrawObj.java
 * For use with the Necklace Illustrator Applet
 * by David Fraser
 * Rochester Institute of Technology
 * May 2005
 *
 */

package Necklace;

import java.awt.Canvas;
import java.awt.Graphics;
import java.awt.Color;
import java.util.Vector;
import Necklace.NecklaceObj;

public class DrawObj extends Canvas{

   Graphics g;
   Vector strings1;
   int startIndex;
   int endIndex;
   boolean tooBig;
   boolean empty;

   //create a new DrawObj
   public DrawObj() {

      g = getGraphics();
      strings1 = new Vector();
      startIndex = 0;
      endIndex = 1;
      tooBig = false;
      empty = true;
   }

   //create a new DrawObj that is passed true/false
   public DrawObj(boolean c){
      g = getGraphics();
      strings1 = new Vector();
      startIndex=0;
      endIndex=0;
      tooBig=true;
      empty=true;
```

```
}

//create a new DrawObj that is passed a vector of strings, the startIndex, endIndex,
and true/false
public DrawObj(Vector vec, int a, int b, boolean c){
   g = getGraphics();
   strings1 = vec;
   startIndex = a;
   endIndex = b;
   tooBig = c;
   empty = false;
   if (strings1.size()==0){
      empty = true;
   }
}

//paint method used to render to screen
public void paint(Graphics g){

   g.setPaintMode();
   g.clearRect(0, 0, 225,350);
   //if the color selection > 14 beads with 4 colors
   if (tooBig){
      g.setColor(Color.BLACK);
      g.drawString("Too large to compute", 40, 20);
   } else {
      //if Vector containing NecklaceObjs is not empty
      if (!empty) {
         //if not last page to draw
         if (strings1.size()>endIndex+1){
            //draw title
            g.drawString("Necklaces "+(startIndex+1)+" through
             "+(endIndex+1),40,20);
            //set initial necklace location
            int xVal = 40;
            int yVal = 29;
            int circleCounter = 1;
            int j = startIndex;
            //draw first two
            while (circleCounter<3) {
               NecklaceObj tempObj = new NecklaceObj();
               tempObj = (NecklaceObj)(strings1.elementAt(j));
               g.setColor(Color.black);
               g.drawOval(xVal,yVal,50,50);
               String str = tempObj.getForStr();
```

```
        for (int i=0; i<str.length(); i++){
            switch(str.charAt(i)){
                case 'r':  g.setColor(Color.red);
                           break;
                case 'b':  g.setColor(Color.blue);
                           break;
                case 'g':  g.setColor(Color.green);
                           break;
                case 'y':  g.setColor(Color.yellow);
                           break;
            }
            int tempX = (int)(xVal+20+(25*Math.cos(i*2*Math.PI/str.length())));
            int tempY = (int)(yVal+20+(25*Math.sin(i*2*Math.PI/str.length())));
            g.fillOval (tempX, tempY, 10, 10);
        }
        xVal = xVal+90;
        circleCounter++;
        j++;
    }
    //reset location
    xVal = 40;
    yVal = 93;
    circleCounter = 1;
    //draw next two
    while (circleCounter<3){
        NecklaceObj tempObj = new NecklaceObj();
        tempObj = (NecklaceObj)(strings1.elementAt(j));
        g.setColor(Color.black);
        g.drawOval(xVal,yVal,50,50);
        String str = tempObj.getForStr();
        for (int i=0; i<str.length(); i++){
            switch(str.charAt(i)){
                case 'r':  g.setColor(Color.red);
                           break;
                case 'b':  g.setColor(Color.blue);
                           break;
                case 'g':  g.setColor(Color.green);
                           break;
                case 'y':  g.setColor(Color.yellow);
                           break;
            }
            int tempX = (int)(xVal+20+(25*Math.cos(i*2*Math.PI/str.length())));
            int tempY = (int)(yVal+20+(25*Math.sin(i*2*Math.PI/str.length())));
            g.fillOval (tempX, tempY, 10, 10);
        }
```

```
    xVal = xVal+90;
    circleCounter++;
    j++;
  }
  //reset location
  xVal = 40;
  yVal = 157;
  circleCounter = 1;
  //draw next two
  while (circleCounter<3){
    NecklaceObj tempObj = new NecklaceObj();
    tempObj = (NecklaceObj)(strings1.elementAt(j));
    g.setColor(Color.black);
    g.drawOval(xVal,yVal,50,50);
    String str = tempObj.getForStr();
    for (int i=0; i<str.length(); i++){
      switch(str.charAt(i)){
        case 'r':  g.setColor(Color.red);
                   break;
        case 'b':  g.setColor(Color.blue);
                   break;
        case 'g':  g.setColor(Color.green);
                   break;
        case 'y':  g.setColor(Color.yellow);
                   break;
      }
      int tempX = (int)(xVal+20+(25*Math.cos(i*2*Math.PI/str.length())));
      int tempY = (int)(yVal+20+(25*Math.sin(i*2*Math.PI/str.length())));
      g.fillOval (tempX, tempY, 10, 10);
    }
    xVal = xVal+90;
    circleCounter++;
    j++;
  }
  //reset location
  xVal = 40;
  yVal = 221;
  circleCounter = 1;
  //draw next two
  while (circleCounter<3){
    NecklaceObj tempObj = new NecklaceObj();
    tempObj = (NecklaceObj)(strings1.elementAt(j));
    g.setColor(Color.black);
    g.drawOval(xVal,yVal,50,50);
    String str = tempObj.getForStr();
```

```
    for (int i=0; i<str.length(); i++){
      switch(str.charAt(i)){
        case 'r':  g.setColor(Color.red);
                   break;
        case 'b':  g.setColor(Color.blue);
                   break;
        case 'g':  g.setColor(Color.green);
                   break;
        case 'y':  g.setColor(Color.yellow);
                   break;
      }
      int tempX = (int)(xVal+20+(25*Math.cos(i*2*Math.PI/str.length())));
      int tempY = (int)(yVal+20+(25*Math.sin(i*2*Math.PI/str.length())));
      g.fillOval (tempX, tempY, 10, 10);
    }
    xVal = xVal+90;
    circleCounter++;
    j++;
  }
  //reset location
  xVal = 40;
  yVal = 285;
  circleCounter = 1;
  //draw next two
  while (circleCounter<3){
    NecklaceObj tempObj = new NecklaceObj();
    tempObj = (NecklaceObj)(strings1.elementAt(j));
    g.setColor(Color.black);
    g.drawOval(xVal,yVal,50,50);
    String str = tempObj.getForStr();
    for (int i=0; i<str.length(); i++){
      switch(str.charAt(i)){
        case 'r':  g.setColor(Color.red);
                   break;
        case 'b':  g.setColor(Color.blue);
                   break;
        case 'g':  g.setColor(Color.green);
                   break;
        case 'y':  g.setColor(Color.yellow);
                   break;
      }
      int tempX = (int)(xVal+20+(25*Math.cos(i*2*Math.PI/str.length())));
      int tempY = (int)(yVal+20+(25*Math.sin(i*2*Math.PI/str.length())));
      g.fillOval (tempX, tempY, 10, 10);
    }
```

```
        xVal = xVal+90;
        circleCounter++;
        j++;
      }
    }
    //if last page to draw
    if (strings1.size()<=endIndex+1){
      g.setColor(Color.BLACK);
      //draw title
      g.drawString("Necklaces "+(startIndex+1)+" through "+strings1.size(),40,20);
      //set initial location
      int xVal = 40;
      int yVal = 29;
      int circleCounter = 1;
      int j = startIndex;
      //draw first two
      while ((circleCounter<3) && (j<strings1.size())){
        NecklaceObj tempObj = new NecklaceObj();
        tempObj = (NecklaceObj)(strings1.elementAt(j));
        g.setColor(Color.black);
        g.drawOval(xVal,yVal,50,50);
        String str = tempObj.getForStr();
        for (int i=0; i<str.length(); i++){
          switch(str.charAt(i)){
            case 'r':  g.setColor(Color.red);
                       break;
            case 'b':  g.setColor(Color.blue);
                       break;
            case 'g':  g.setColor(Color.green);
                       break;
            case 'y':  g.setColor(Color.yellow);
                       break;
          }
          int tempX = (int)(xVal+20+(25*Math.cos(i*2*Math.PI/str.length())));
          int tempY = (int)(yVal+20+(25*Math.sin(i*2*Math.PI/str.length())));
          g.fillOval (tempX, tempY, 10, 10);
        }
        xVal = xVal+90;
        circleCounter++;
        j++;
      }
      //reset location
      xVal = 40;
      yVal = 93;
      circleCounter = 1;
```

```
//draw next two
while ((circleCounter<3) && (j<strings1.size())){
   NecklaceObj tempObj = new NecklaceObj();
   tempObj = (NecklaceObj)(strings1.elementAt(j));
   g.setColor(Color.black);
   g.drawOval(xVal,yVal,50,50);
   String str = tempObj.getForStr();
   for (int i=0; i<str.length(); i++){
      switch(str.charAt(i)){
         case 'r':  g.setColor(Color.red);
                    break;
         case 'b':  g.setColor(Color.blue);
                    break;
         case 'g':  g.setColor(Color.green);
                    break;
         case 'y':  g.setColor(Color.yellow);
                    break;
      }
      int tempX = (int)(xVal+20+(25*Math.cos(i*2*Math.PI/str.length())));
      int tempY = (int)(yVal+20+(25*Math.sin(i*2*Math.PI/str.length())));
      g.fillOval (tempX, tempY, 10, 10);
   }
   xVal = xVal+90;
   circleCounter++;
   j++;
}
//reset location
xVal = 40;
yVal = 157;
circleCounter = 1;
//draw next two
while ((circleCounter<3) && (j<strings1.size())){
   NecklaceObj tempObj = new NecklaceObj();
   tempObj = (NecklaceObj)(strings1.elementAt(j));
   g.setColor(Color.black);
   g.drawOval(xVal,yVal,50,50);
   String str = tempObj.getForStr();
   for (int i=0; i<str.length(); i++){
      switch(str.charAt(i)){
         case 'r':  g.setColor(Color.red);
                    break;
         case 'b':  g.setColor(Color.blue);
                    break;
         case 'g':  g.setColor(Color.green);
                    break;
```

```
            case 'y':  g.setColor(Color.yellow);
                       break;
         }
         int tempX = (int)(xVal+20+(25*Math.cos(i*2*Math.PI/str.length())));
         int tempY = (int)(yVal+20+(25*Math.sin(i*2*Math.PI/str.length())));
         g.fillOval (tempX, tempY, 10, 10);
      }
      xVal = xVal+90;
      circleCounter++;
      j++;
   }
   //reset location
   xVal = 40;
   yVal = 221;
   circleCounter = 1;
   //draw next two
   while ((circleCounter<3) && (j<strings1.size())){
      NecklaceObj tempObj = new NecklaceObj();
      tempObj = (NecklaceObj)(strings1.elementAt(j));
      g.setColor(Color.black);
      g.drawOval(xVal,yVal,50,50);
      String str = tempObj.getForStr();
      for (int i=0; i<str.length(); i++){
         switch(str.charAt(i)){
            case 'r':  g.setColor(Color.red);
                       break;
            case 'b':  g.setColor(Color.blue);
                       break;
            case 'g':  g.setColor(Color.green);
                       break;
            case 'y':  g.setColor(Color.yellow);
                       break;
         }
         int tempX = (int)(xVal+20+(25*Math.cos(i*2*Math.PI/str.length())));
         int tempY = (int)(yVal+20+(25*Math.sin(i*2*Math.PI/str.length())));
         g.fillOval (tempX, tempY, 10, 10);
      }
      xVal = xVal+90;
      circleCounter++;
      j++;
   }
   //reset location
   xVal = 40;
   yVal = 285;
   circleCounter = 1;
```

```
            //draw next two
            while ((circleCounter<3) && (j<strings1.size())){
               NecklaceObj tempObj = new NecklaceObj();
               tempObj = (NecklaceObj)(strings1.elementAt(j));
               g.setColor(Color.black);
               g.drawOval(xVal,yVal,50,50);
               String str = tempObj.getForStr();
               for (int i=0; i<str.length(); i++){
                  switch(str.charAt(i)){
                     case 'r':  g.setColor(Color.red);
                              break;
                     case 'b':  g.setColor(Color.blue);
                              break;
                     case 'g':  g.setColor(Color.green);
                              break;
                     case 'y':  g.setColor(Color.yellow);
                              break;
                  }
                  int tempX = (int)(xVal+20+(25*Math.cos(i*2*Math.PI/str.length())));
                  int tempY = (int)(yVal+20+(25*Math.sin(i*2*Math.PI/str.length())));
                  g.fillOval (tempX, tempY, 10, 10);
               }
               xVal = xVal+90;
               circleCounter++;
               j++;
            }
         }
      }else {
         //if Vector containing NecklaceObjs is empty
         g.drawString("Nothing selected",40,20 );
      }
   }
   //free system resources
   g.dispose();
   }

}
```

```
/*
 * Necklace.java
 * For use with the Necklace Illustrator Applet
 * by David Fraser
 * Rochester Institute of Technology
 * May 2005
 *
 */

import java.awt.*;
import java.awt.FlowLayout;
import java.awt.Color;
import java.awt.Graphics;
import java.applet.*;
import java.util.*;
import javax.swing.JComboBox;
import javax.swing.JScrollPane;
import javax.swing.text.*;
import java.awt.event.*;
import java.lang.*;
import Necklace.NecklaceCompute;
import Necklace.NecklaceObj;
import Necklace.CycleIndex;
import Necklace.CycleIndexList;
import Necklace.CycleIndexObj;
import Necklace.PatternInventory;
import Necklace.PatternInventoryObj;
import Necklace.DrawObj;
import Necklace.CIDraw;
import Necklace.PIDraw;
import java.math.*;

public class Necklace extends Applet implements ActionListener{

   private Label title;
   private JComboBox blueBox;
   private Label blueLabel;
   private Button drawButton;
   private JComboBox greenBox;
   private Label greenLabel;
   private Label illustratorTitle;
   private Panel outPanel;
   private JComboBox redBox;
   private Label redLabel;
   private JComboBox yellowBox;
```

```
private Label yellowLabel;
private Vector strings1;
private ScrollPane cycleSP;
private ScrollPane patternSP;
private ScrollPane drawSP;
private ScrollPane entrySP;
private Panel entryPanel;
private DrawObj drawPanel;
private CIDraw cyclePanel;
private PIDraw patternPanel;
private int startIndex = 0;
private int endIndex = 9;
private boolean listBuilt = false;
private Button newButton;
private boolean newBFlag = false;
private int totalBeads = 0;
private Panel necklacePanel;
private boolean tooBig = false;
private Vector emptyVec;
private int max=0;

//create new Necklace
public Necklace() {
  begin();
}

private void begin(){

   Vector beadNumbers = new Vector();
   for (int i=0; i<6; i++) {
      String tempString = new String(""+i+"");
      beadNumbers.add(tempString);
   }
   //create pane for input and components
   entrySP = new ScrollPane(2);
   entryPanel = new Panel();
   entryPanel.setLayout(null);
   entryPanel.setBackground(Color.WHITE);
   title = new Label("The Necklace Illustrator", Label.CENTER);
   title.setBackground(Color.WHITE);
   title.setFont(new Font("Dialog", Font.ITALIC, 24));
   redLabel = new Label("Red Beads", Label.RIGHT);
   redLabel.setBackground(Color.WHITE);
   blueLabel = new Label("Blue Beads", Label.RIGHT);
   blueLabel.setBackground(Color.WHITE);
```

```
greenLabel = new Label("Green Beads", Label.RIGHT);
greenLabel.setBackground(Color.WHITE);
yellowLabel = new Label("Yellow Beads", Label.RIGHT);
yellowLabel.setBackground(Color.WHITE);
redBox = new JComboBox(beadNumbers);
blueBox = new JComboBox(beadNumbers);
greenBox = new JComboBox(beadNumbers);
yellowBox = new JComboBox(beadNumbers);
drawButton = new Button("Draw first 10");
//create panes for output and components
cycleSP = new ScrollPane(ScrollPane.SCROLLBARS_NEVER);
patternSP = new ScrollPane(ScrollPane.SCROLLBARS_AS_NEEDED);
drawSP = new ScrollPane(ScrollPane.SCROLLBARS_NEVER);
emptyVec = new Vector();
cyclePanel = new CIDraw(emptyVec);
cyclePanel.setBackground(Color.WHITE);
cyclePanel.setBounds(0,0,320,70);
patternPanel = new PIDraw(emptyVec,0,0,0,0);
patternPanel.setBackground(Color.WHITE);
patternPanel.setBounds(0,0,600,40);
drawPanel = new DrawObj();
drawPanel.setBounds(0, 0, 195, 300);
drawSP.add(drawPanel);
drawSP.setBackground(Color.WHITE);
setLayout(null);
//add pane components
entryPanel.add(title);
entryPanel.add(redLabel);
entryPanel.add(blueLabel);
entryPanel.add(greenLabel);
entryPanel.add(yellowLabel);
entryPanel.add(redBox);
entryPanel.add(blueBox);
entryPanel.add(greenBox);
entryPanel.add(yellowBox);
add(drawButton);
add(entrySP);
add(cycleSP);
add(patternSP);
add(drawSP);
entrySP.add(entryPanel);
entryPanel.setBounds(0,0,430,200);
entrySP.setBounds(0, 0, 430, 275);
cycleSP.setBounds(0, 275, 430, 75);
patternSP.setBounds(0, 350, 650, 150);
```

```
    drawSP.setBounds(430, 0, 220, 350);
    title.setBounds(0, 20, 430, 40);
    redLabel.setBounds(40, 70, 90, 25);
    blueLabel.setBounds(40, 100, 90, 25);
    greenLabel.setBounds(220, 70, 90, 25);
    yellowLabel.setBounds(220, 100, 90, 25);
    redBox.setBounds(140, 70, 50, 25);
    blueBox.setBounds(140, 100, 50, 25);
    greenBox.setBounds(320, 70, 50, 25);
    yellowBox.setBounds(320, 100, 50, 25);
    redBox.addActionListener(this);
    blueBox.addActionListener(this);
    greenBox.addActionListener(this);
    yellowBox.addActionListener(this);
    drawButton.setBounds(80, 140, 110, 25);
    drawButton.addActionListener(this);
    cycleSP.add(cyclePanel);
    patternSP.add(patternPanel);
}

public void actionPerformed(ActionEvent event){

    int red = redBox.getSelectedIndex();
    int blue = blueBox.getSelectedIndex();
    int green = greenBox.getSelectedIndex();
    int yellow = yellowBox.getSelectedIndex();
    //if a color box is changed
    if((event.getSource()==redBox)||(event.getSource()==blueBox)||
    (event.getSource()==greenBox)||(event.getSource()==yellowBox)) {
        totalBeads = red + blue + green + yellow;
        //clear Cycle Index and Pattern Inventory panes
        cycleSP.remove(cyclePanel);
        patternSP.removeAll();
        //recalculate Cycle Index
        CycleIndex cyIdx = new CycleIndex(totalBeads);
        Vector cyIdxVec = cyIdx.getCIVector();
        cyclePanel = new CIDraw(cyIdxVec);
        cyclePanel.setBackground(Color.WHITE);
        Integer convInt = new Integer(0);
        //display Cycle Index
        cycleSP.add(cyclePanel);
        cycleSP.doLayout();
        //recalculate Pattern Inventory
        PatternInventory pattInv = new PatternInventory(cyIdxVec, red, blue, green,
        yellow);
```

```
    Vector pattVec = pattInv.getVec();
    patternPanel = new PIDraw(pattVec, red, blue, green, yellow );
    patternPanel.setSize(600,patternPanel.getHeight()+20);
    patternPanel.setBackground(Color.WHITE);
    //display Pattern Inventory
    patternSP.add(patternPanel);
    patternSP.doLayout();
  }
  //if a "draw" button is clicked
  if ((event.getSource()==drawButton)||(event.getSource()==newButton)) {
    NecklaceCompute neckComp = new NecklaceCompute();
    //if "new" button
    if (event.getSource()==newButton) {
      entryPanel.remove(newButton);
      startIndex=0;
      endIndex=9;
      listBuilt=false;
      max=patternPanel.getMax();
    }
    //determine if selection too large to compute
    if (!listBuilt){
      tooBig = false;
      max=patternPanel.getMax();
      int tempColors=0;
      if (red!=0) {
        tempColors++;
      }
      if (blue!=0){
        tempColors++;
      }
      if (green!=0){
        tempColors++;
      }
      if (yellow!=0){
        tempColors++;
      }
      int tempTotal=red+blue+green+yellow;
      if ((tempTotal>14)&&(tempColors==4)){
        tooBig=true;
      } else {
        //generate permutations
        strings1 = neckComp.getPermutations(red, blue, green, yellow);
        listBuilt=true;
      }
    }
```

```
//if not too large to compute
if (!tooBig){
   //determine unique necklaces and draw to screen
   strings1 = neckComp.uniqueNecklaces(strings1, startIndex, endIndex);
   //if more strings need to be checked
   if (strings1.size()>endIndex+1){
      drawPanel = new DrawObj(strings1, startIndex, endIndex, tooBig);
      drawPanel.setBounds(0, 0, 195, 360);
      drawSP.add(drawPanel);
      drawButton.setLabel("Draw next 10");
      if(newBFlag){
         entryPanel.remove(newButton);
      }
      newButton = new Button("New necklace");
      entryPanel.add(newButton);
      newBFlag = true;
      newButton.setBounds(260, 140, 110, 25);
      newButton.addActionListener(this);
      //if maximum number of unique necklaces are already shown
      if (startIndex+10>=max){
         entryPanel.remove(newButton);
         listBuilt=false;
         drawButton.setLabel("Draw first 10");
         startIndex=-10;
         endIndex=-1;
         newBFlag=false;
      }
      startIndex=startIndex+10;
      endIndex=endIndex+10;
   //if no more strings need to be checked
   } else {
      drawPanel = new DrawObj(strings1, startIndex, endIndex, tooBig);
      drawPanel.setBounds(0, 0, 195, 360);
      drawSP.add(drawPanel);
      listBuilt=false;
      drawButton.setLabel("Draw first 10");
      startIndex=0;
      endIndex=9;
      if (newBFlag){
         entryPanel.remove(newButton);
         newBFlag=false;
      }
   }
//if too large to compute
} else {
```

```
                drawPanel = new DrawObj(true);
                drawButton.setLabel("Draw first 10");
                drawPanel.setBounds(0, 0, 195, 300);
                drawSP.add(drawPanel);
            }
        }
    }

}
```

```
/*
 * NecklaceCompute.java
 * For use with the Necklace Illustrator Applet
 * by David Fraser
 * Rochester Institute of Technology
 * May 2005
 *
 */

package Necklace;

import java.util.Vector;
import Necklace.NecklaceObj;

public class NecklaceCompute {

   private char leastColor = 'r';
   private Vector strings;
   private int colorTotal = 0;

   //create a new NecklaceCompute
   public NecklaceCompute() {
   }

   public Vector getPermutations(int red, int blue, int green, int yellow) throws
OutOfMemoryError{

      int max = 5;
      colorTotal=0;
      //determine number of colors
      if (red!=0) {
         colorTotal++;
      }
      if (blue!=0) {
         colorTotal++;
      }
      if (green!=0) {
         colorTotal++;
      }
      if (yellow!=0) {
         colorTotal++;
      }
      //determine smallest color
      if ((yellow <= max) && (yellow != 0)) {
         leastColor='y';
```

```
            max = yellow;
        }
        if ((green <= max) && (green != 0)) {
            leastColor='g';
            max = green;
        }
        if ((blue <= max) && (blue != 0)) {
            leastColor='b';
            max = blue;
        }
        if ((red <= max) && (red != 0)) {
            leastColor='r';
            max = red;
        }
        strings = new Vector();
        String permOne = new String();
        //build string of r-b-g-y
        for (int i=0; i<red; i++) {
            permOne = permOne.concat("r");
        }
        for (int i=0; i<blue; i++) {
            permOne = permOne.concat("b");
        }
        for (int i=0; i<green; i++) {
            permOne = permOne.concat("g");
        }
        for (int i=0; i<yellow; i++) {
            permOne = permOne.concat("y");
        }
        //begin building permutations
        permutationGenerator1("", permOne);
        return strings;
    }

    private void permutationGenerator1(String prefix, String suffix) {

        String newPrefix="", newSuffix="";
        int numOfChars = suffix.length();
        int numOfRed = 0;
        int lastRed = 0;
        int numOfBlue = 0;
        int lastBlue = 0;
        int numOfGreen = 0;
        int lastGreen = 0;
        int numOfYellow = 0;
```

```
int lastYellow = 0;
//compute number, last location of each color
for (int i=0; i<numOfChars; i++) {
   if (suffix.charAt(i)=='r') {
     numOfRed++;
     lastRed=i;
   }
   if (suffix.charAt(i)=='b') {
     numOfBlue++;
     lastBlue=i;
   }
   if (suffix.charAt(i)=='g') {
     numOfGreen++;
     lastGreen=i;
   }
   if (suffix.charAt(i)=='y') {
     numOfYellow++;
     lastYellow=i;
   }
}
//if one character left
if (numOfChars==1) {
   newPrefix = prefix + suffix;
   //if only one color used
   if (colorTotal==1){
      NecklaceObj neckObj = new NecklaceObj(newPrefix);
      strings.add(neckObj);
   }
   if (prefix.length()!=0){
      if ((prefix.charAt(0)==leastColor) && (prefix.charAt(0)!=suffix.charAt(0))) {
         NecklaceObj neckObj = new NecklaceObj(newPrefix);
         strings.add(neckObj);
      }
   }
//if multiple characters left
} else {
   //if smallest color used is red
   if ((numOfRed != 0)&&(leastColor=='r')) {
      if (lastRed==0) {
         newSuffix = suffix.substring(1,numOfChars);
      }
      if (lastRed==numOfChars-1) {
         newSuffix = suffix.substring(0,numOfChars-1);
      }
      if ((lastRed!=0)&&(lastRed!=numOfChars-1)) {
```

```
        newSuffix = suffix.substring(0,lastRed) +
        suffix.substring(lastRed+1,numOfChars);
    }
    newPrefix = prefix + "r";
    permutationGenerator(newPrefix, newSuffix);
}
//if smallest color used is blue
if ((numOfBlue != 0)&&(leastColor=='b')) {
    if (lastBlue==0) {
        newSuffix = suffix.substring(1,numOfChars);
    }
    if (lastBlue==numOfChars-1) {
        newSuffix = suffix.substring(0,numOfChars-1);
    }
    if ((lastBlue!=0)&&(lastBlue!=numOfChars-1)) {
        newSuffix = suffix.substring(0,lastBlue) +
        suffix.substring(lastBlue+1,numOfChars);
    }
    newPrefix = prefix + "b";
    permutationGenerator(newPrefix, newSuffix);
}
//if smallest color used is green
if ((numOfGreen != 0)&&(leastColor=='g')) {
    if (lastGreen==0) {
        newSuffix = suffix.substring(1,numOfChars);
    }
    if (lastGreen==numOfChars-1) {
        newSuffix = suffix.substring(0,numOfChars-1);
    }
    if ((lastGreen!=0)&&(lastGreen!=numOfChars-1)) {
      newSuffix = suffix.substring(0,lastGreen) +
      suffix.substring(lastGreen+1,numOfChars);
    }
    newPrefix = prefix + "g";
    permutationGenerator(newPrefix, newSuffix);
}
//if smallest color used is yellow
if ((numOfYellow != 0)&&(leastColor=='y')) {
    if (lastYellow==0) {
        newSuffix = suffix.substring(1,numOfChars);
    }
    if (lastYellow==numOfChars-1) {
        newSuffix = suffix.substring(0,numOfChars-1);
    }
    if ((lastYellow!=0)&&(lastYellow!=numOfChars-1)) {
```

```
            newSuffix = suffix.substring(0,lastYellow) +
            suffix.substring(lastYellow+1,numOfChars);
        }
        newPrefix = prefix + "y";
        permutationGenerator(newPrefix, newSuffix);
      }
    }

}

private void permutationGenerator(String prefix, String suffix) {

    String newPrefix="", newSuffix="";
    int numOfChars = suffix.length();
    int numOfRed = 0;
    int lastRed = 0;
    int numOfBlue = 0;
    int lastBlue = 0;
    int numOfGreen = 0;
    int lastGreen = 0;
    int numOfYellow = 0;
    int lastYellow = 0;
    //compute number, last location of each color
    for (int i=0; i<numOfChars; i++) {
      if (suffix.charAt(i)=='r') {
        numOfRed++;
        lastRed=i;
      }
      if (suffix.charAt(i)=='b') {
        numOfBlue++;
        lastBlue=i;
      }
      if (suffix.charAt(i)=='g') {
        numOfGreen++;
        lastGreen=i;
      }
      if (suffix.charAt(i)=='y') {
        numOfYellow++;
        lastYellow=i;
      }
    }
    //if one character left
    if (numOfChars==1) {
      newPrefix = prefix + suffix;
      if (colorTotal==1){
```

```
            NecklaceObj neckObj = new NecklaceObj(newPrefix);
            strings.add(neckObj);
        }
        if (prefix.length()!=0){
            if ((prefix.charAt(0)==leastColor) && (prefix.charAt(0)!=suffix.charAt(0))) {
                NecklaceObj neckObj = new NecklaceObj(newPrefix);
                strings.add(neckObj);
            }
        }
    //if multiple characters left
    } else {
        //if red remains, add r to prefix
        if (numOfRed != 0) {
            if (lastRed==0) {
                newSuffix = suffix.substring(1,numOfChars);
            }
            if (lastRed==numOfChars-1) {
                newSuffix = suffix.substring(0,numOfChars-1);
            }
            if ((lastRed!=0)&&(lastRed!=numOfChars-1)) {
                newSuffix = suffix.substring(0,lastRed) +
                suffix.substring(lastRed+1,numOfChars);
            }
            newPrefix = prefix + "r";
            permutationGenerator(newPrefix, newSuffix);
        }
        //if blue remains, add b to prefix
        if (numOfBlue != 0) {
            if (lastBlue==0) {
                newSuffix = suffix.substring(1,numOfChars);
            }
            if (lastBlue==numOfChars-1) {
                newSuffix = suffix.substring(0,numOfChars-1);
            }
            if ((lastBlue!=0)&&(lastBlue!=numOfChars-1)) {
                newSuffix = suffix.substring(0,lastBlue) +
                suffix.substring(lastBlue+1,numOfChars);
            }
            newPrefix = prefix + "b";
            permutationGenerator(newPrefix, newSuffix);
        }
        //if green remains, add g to prefix
        if (numOfGreen != 0) {
            if (lastGreen==0) {
                newSuffix = suffix.substring(1,numOfChars);
```

```
            }
            if (lastGreen==numOfChars-1) {
               newSuffix = suffix.substring(0,numOfChars-1);
            }
            if ((lastGreen!=0)&&(lastGreen!=numOfChars-1)) {
               newSuffix = suffix.substring(0,lastGreen) +
               suffix.substring(lastGreen+1,numOfChars);
            }
            newPrefix = prefix + "g";
            permutationGenerator(newPrefix, newSuffix);
         }
         //if yellow remains, add y to prefix
         if (numOfYellow != 0) {
            if (lastYellow==0) {
               newSuffix = suffix.substring(1,numOfChars);
            }
            if (lastYellow==numOfChars-1) {
               newSuffix = suffix.substring(0,numOfChars-1);
            }
            if ((lastYellow!=0)&&(lastYellow!=numOfChars-1)) {
               newSuffix = suffix.substring(0,lastYellow) +
               suffix.substring(lastYellow+1,numOfChars);
            }
            newPrefix = prefix + "y";
            permutationGenerator(newPrefix, newSuffix);
         }
      }

}

public Vector uniqueNecklaces(Vector strings1, int startIndex, int endIndex) {

   boolean duplicate = false;
   int i = startIndex;
   if (i==0){
     i = 1;
   }
   int j = 0;
   int counter = 0;
   String forString1 = "";
   String forString2 = "";
   long forNum1 = 1;
   long forNum2 = 1;
   long backNum1 = 1;
   //while not at the end of the Vector
```

```
while ((i<=endIndex)&&(i<strings1.size())) {
  duplicate = false;
  while (!duplicate && j<i) {
    //compare objects at locations i and j
    NecklaceObj neckObj1 = (NecklaceObj)strings1.elementAt(i);
    NecklaceObj neckObj2 = (NecklaceObj)strings1.elementAt(j);
    forNum1 = neckObj1.getForNum();
    backNum1 = neckObj1.getBackNum();
    forNum2 = neckObj2.getForNum();
    //if forward numbers match
    if (!duplicate && forNum1==forNum2) {
      counter = 0;
      String checkStr1 = neckObj1.getForStr();
      String checkStr2 = neckObj2.getForStr();
      //compare strings with rotation
      while (!duplicate && counter<checkStr1.length()) {
        if (checkStr1.equals(checkStr2)) {
          strings1.remove(i);
          duplicate = true;
        } else {
          checkStr1=checkStr1.substring(1,checkStr1.length())
          +checkStr1.charAt(0);
          counter++;
        }
      }
    }
    //if forawd number matches back number
    if (!duplicate && backNum1==forNum2) {
      counter = 0;
      String checkStr1 = neckObj1.getBackStr();
      String checkStr2 = neckObj2.getForStr();
      //compare strings with rotation
      while (!duplicate && counter<checkStr1.length()) {
        if (checkStr1.equals(checkStr2)) {
          strings1.remove(i);
          duplicate = true;
        } else {
          checkStr1 = checkStr1.substring(1,checkStr1.length())
          +checkStr1.charAt(0);
          counter++;
        }
      }
    }
    if (!duplicate) {
      j++;
```

```
            } else {
                j=0;
            }
        }
        if (!duplicate) {
            i++;
            j=0;
        }
    }
    return strings1;

  }

}
```

```
/*
 * NecklaceObj.java
 * For use with the Necklace Illustrator Applet
 * by David Fraser
 * Rochester Institute of Technology
 * May 2005
 *
 */

package Necklace;

public class NecklaceObj {

  private String forString = "";
  private String backString = "";
  private long forNum = 1;
  private long backNum = 1;

    //create a new NecklaceObj
    public NecklaceObj() {
    }

    //create a new NecklaceObj that is passed a string
    public NecklaceObj(String str) {

        forString = str;
        int m = str.length()-1;
        for (int i=0; i<str.length(); i++) {
            backString = backString+forString.charAt(m);
            m--;
        }
        for (int j=0; j<str.length()-1; j++) {
            forNum = forNum * generateNum(forString.charAt(j), forString.charAt(j+1));
        }
        forNum = forNum * generateNum(forString.charAt(forString.length()-1),
        forString.charAt(0));
        for (int k=0; k<str.length()-1; k++) {
            backNum = backNum * generateNum(backString.charAt(k),
            backString.charAt(k+1));
        }
        backNum = backNum * generateNum(backString.charAt(backString.length()-1),
        backString.charAt(0));

  }
```

```
//return the forward string
public String getForStr() {
   return forString;
}

//return the backward string
public String getBackStr() {
   return backString;
}

//return the forward number
public long getForNum() {
   return forNum;
}

//return the backward number
public long getBackNum() {
   return backNum;
}

//set the forward string
public void setForStr(String str){
   forString = str;
}

//get prime number to multiply forward/backward number
private long generateNum(char a, char b) {

   //this returns an int value depending on current and next color
   if (a=='r' && b=='r') {
    return 1;
   }
   if (a=='r' && b=='b') {
    return 2;
   }
   if (a=='r' && b=='g') {
    return 3;
   }
   if (a=='r' && b=='y') {
    return 5;
   }
   if (a=='b' && b=='r') {
    return 7;
   }
   if (a=='b' && b=='b') {
```

```
            return 11;
          }
          if (a=='b' && b=='g') {
            return 13;
          }
          if (a=='b' && b=='y') {
            return 17;
          }
          if (a=='g' && b=='r') {
            return 19;
          }
          if (a=='g' && b=='b') {
            return 23;
          }
          if (a=='g' && b=='g') {
            return 29;
          }
          if (a=='g' && b=='y') {
            return 31;
          }
          if (a=='y' && b=='r') {
            return 37;
          }
          if (a=='y' && b=='b') {
            return 41;
          }
          if (a=='y' && b=='g') {
            return 43;
          }
          if (a=='y' && b=='y') {
            return 47;
          }
          return 1;

      }

}
```

```
/*
 * PatternInventory.java
 * For use with the Necklace Illustrator Applet
 * by David Fraser
 * Rochester Institute of Technology
 * May 2005
 *
 */

package Necklace;

import java.util.Vector;
import java.lang.Math.*;
import Necklace.CycleIndexList;
import Necklace.CycleIndexObj;

public class PatternInventory {

   int total;
   Vector ciVector;
   Vector compVector;
   Vector colors;
   Vector storeVec1;
   Vector storeVec2;
   Vector storeVec3;
   Vector storeVec4;

   //create a new PatternInventory
   public PatternInventory() {
   }

   //create a new PatternInventory that is passed the Cycle Index Vector and number of
  each color
   public PatternInventory(Vector vec, int r, int b, int g, int y){

      ciVector = vec;
      total=r+b+g+y;
      String colorStr = new String();
      colors = new Vector();
      compVector = new Vector();
      if (r!=0){
         colorStr = "r";
         colors.add(colorStr);
      }
      if (b!=0){
```

```
            colorStr = "b";
            colors.add(colorStr);
        }
        if (g!=0){
            colorStr = "g";
            colors.add(colorStr);
        }
        if (y!=0){
            colorStr = "y";
            colors.add(colorStr);
        }
        if (total==1){
            computeOneBead();
        }
        if (total==2){
            computeTwoBead();
        }
        if (total>2){
            if (colors.size()==1){
                computeOneColor();
            }
            if (colors.size()==2){
                computeTwoColor();
            }
            if (colors.size()==3){
                computeThreeColor();
            }
            if (colors.size()==4){
                computeFourColor();
            }
        }

    }

    //if there is only one bead
    private void computeOneBead(){

        PatternInventoryObj piObj = new PatternInventoryObj();
        String tempStr = (String)colors.elementAt(0);
        if (tempStr=="r"){
            piObj.setRExp(1);
        }
        if (tempStr=="b"){
            piObj.setBExp(1);
        }
```

```
    if (tempStr=="g"){
      piObj.setGExp(1);
    }
    if (tempStr=="y"){
      piObj.setYExp(1);
    }
    compVector.add(piObj);

}

//if there are two beads
private void computeTwoBead(){

    //if only one color
    if (colors.size()==1){
      PatternInventoryObj piObj = new PatternInventoryObj();
      String tempStr = (String)colors.elementAt(0);
      if (tempStr=="r"){
        piObj = new PatternInventoryObj(1,2,0,0,0);
      }
      if (tempStr=="b"){
        piObj = new PatternInventoryObj(1,0,2,0,0);
      }
      if (tempStr=="g"){
        piObj = new PatternInventoryObj(1,0,0,2,0);
      }
      if (tempStr=="y"){
        piObj = new PatternInventoryObj(1,0,0,0,2);
      }
      compVector.add(piObj);
    //if there are two colors
    } else{
      //based on the two colors, add terms
      PatternInventoryObj piObj0 = new PatternInventoryObj();
      PatternInventoryObj piObj1 = new PatternInventoryObj();
      PatternInventoryObj piObj2 = new PatternInventoryObj();
      String tempStr0 = (String)colors.elementAt(0);
      String tempStr1 = (String)colors.elementAt(1);
      if ((tempStr0=="r")&&(tempStr1=="b")){
        piObj0 = new PatternInventoryObj(1,2,0,0,0);
        piObj1 = new PatternInventoryObj(1,1,1,0,0);
        piObj2 = new PatternInventoryObj(1,0,2,0,0);
        compVector.add(piObj0);
        compVector.add(piObj1);
        compVector.add(piObj2);
```

```
        }
        if ((tempStr0=="r")&&(tempStr1=="g")){
            piObj0 = new PatternInventoryObj(1,2,0,0,0);
            piObj1 = new PatternInventoryObj(1,1,0,1,0);
            piObj2 = new PatternInventoryObj(1,0,0,2,0);
            compVector.add(piObj0);
            compVector.add(piObj1);
            compVector.add(piObj2);
        }
        if ((tempStr0=="r")&&(tempStr1=="y")){
            piObj0 = new PatternInventoryObj(1,2,0,0,0);
            piObj1 = new PatternInventoryObj(1,1,0,0,1);
            piObj2 = new PatternInventoryObj(1,0,0,0,2);
            compVector.add(piObj0);
            compVector.add(piObj1);
            compVector.add(piObj2);
        }
        if ((tempStr0=="b")&&(tempStr1=="g")){
            piObj0 = new PatternInventoryObj(1,0,2,0,0);
            piObj1 = new PatternInventoryObj(1,0,1,1,0);
            piObj2 = new PatternInventoryObj(1,0,0,2,0);
            compVector.add(piObj0);
            compVector.add(piObj1);
            compVector.add(piObj2);
        }
        if ((tempStr0=="b")&&(tempStr1=="y")){
            piObj0 = new PatternInventoryObj(1,0,2,0,0);
            piObj1 = new PatternInventoryObj(1,0,1,0,1);
            piObj2 = new PatternInventoryObj(1,0,0,0,2);
            compVector.add(piObj0);
            compVector.add(piObj1);
            compVector.add(piObj2);
        }
        if ((tempStr0=="g")&&(tempStr1=="y")){
            piObj0 = new PatternInventoryObj(1,0,0,2,0);
            piObj1 = new PatternInventoryObj(1,0,0,1,1);
            piObj2 = new PatternInventoryObj(1,0,0,0,2);
            compVector.add(piObj0);
            compVector.add(piObj1);
            compVector.add(piObj2);
        }
    }

}
```

```
//multiply two Vectors of Pattern Inventory terms
private Vector multiply(Vector vec1, Vector vec2){

   Vector tempVector = new Vector();
   for (int i=0; i<vec2.size(); i++){
      PatternInventoryObj tempObj2=(PatternInventoryObj)vec2.elementAt(i);
      for (int j=0; j<vec1.size(); j++){
         PatternInventoryObj tempObj1=(PatternInventoryObj)vec1.elementAt(j);
         int newR = tempObj1.getRExp()+tempObj2.getRExp();
         int newB = tempObj1.getBExp()+tempObj2.getBExp();
         int newG = tempObj1.getGExp()+tempObj2.getGExp();
         int newY = tempObj1.getYExp()+tempObj2.getYExp();
         long tempCoeff = tempObj1.getCoeff()*tempObj2.getCoeff();
         PatternInventoryObj tempObj3 = new PatternInventoryObj
         (tempCoeff,newR,newB,newG,newY);
         tempVector.add(tempObj3);
      }
   }
   return tempVector;

}

//add terms together if their exponents match
private Vector groupTerms(Vector vec){

   for (int i=0; i<vec.size(); i++){
      int j=i+1;
      PatternInventoryObj tempObj1 = (PatternInventoryObj)vec.elementAt(i);
      while (j<vec.size()){
         PatternInventoryObj tempObj2 = (PatternInventoryObj)vec.elementAt(j);
         if ((tempObj1.getRExp()==tempObj2.getRExp())&&
         (tempObj1.getBExp()==tempObj2.getBExp())&&(tempObj1.getGExp()==
          tempObj2.getGExp())&&(tempObj1.getYExp()==tempObj2.getYExp())){
            tempObj1.setCoeff(tempObj1.getCoeff()+tempObj2.getCoeff());
            vec.set(i, tempObj1);
            vec.remove(j);
         } else {
            j++;
         }
      }
   }
   return vec;

}
```

```
//return Pattern Inventory Vector
public Vector getVec(){
   return compVector;
}

//compute a factorial
private float factorial(float f){

   if ((f==1)||(f==0)){
      return 1;
   } else {
      return f*factorial(f-1);
   }

}

//if there is only one color
private void computeOneColor(){

   compVector = new Vector();
   String tempStr = (String)colors.elementAt(0);
   if (tempStr=="r"){
      PatternInventoryObj piObj = new PatternInventoryObj(1,total,0,0,0);
      compVector.add(piObj);
   }
   if (tempStr=="b"){
      PatternInventoryObj piObj = new PatternInventoryObj(1,0,total,0,0);
      compVector.add(piObj);
   }
   if (tempStr=="g"){
      PatternInventoryObj piObj = new PatternInventoryObj(1,0,0,total,0);
      compVector.add(piObj);
   }
   if (tempStr=="y"){
      PatternInventoryObj piObj = new PatternInventoryObj(1,0,0,0,total);
      compVector.add(piObj);
   }

}

//if there are two colors
private void computeTwoColor(){

   compVector = new Vector();
   String tempStr0 = (String)colors.elementAt(0);
```

```
String tempStr1 = (String)colors.elementAt(1);
CycleIndexList ciList = new CycleIndexList();
Vector tempVec = new Vector();
//expand for each Cycle Index term but last, add according to colors used
for(int i=0; i<ciVector.size()-1; i++){
   ciList = (CycleIndexList)ciVector.elementAt(i);
   tempVec = ciList.getCycleIndexEntry();
   CycleIndexObj ciObj = (CycleIndexObj)tempVec.elementAt(0);
   for (int j=0; j<=ciObj.getSup(); j++){
      int k=ciObj.getSup()-j;
      float tempCoeff =
     (factorial((float)(j+k)))/((factorial((float)j))*(factorial((float)k)));
      if ((tempStr0=="r")&&(tempStr1=="b")){
         PatternInventoryObj piObj = new
         PatternInventoryObj((long)tempCoeff*ciList.getCoeff(),
         j*ciObj.getSub(),k*ciObj.getSub(),0,0);
         compVector.add(piObj);
      }
      if ((tempStr0=="r")&&(tempStr1=="g")){
         PatternInventoryObj piObj = new
         PatternInventoryObj((long)tempCoeff*ciList.getCoeff(),
         j*ciObj.getSub(),0,k*ciObj.getSub(),0);
         compVector.add(piObj);
      }
      if ((tempStr0=="r")&&(tempStr1=="y")){
         PatternInventoryObj piObj = new
         PatternInventoryObj((long)tempCoeff*ciList.getCoeff(),
         j*ciObj.getSub(),0,0,k*ciObj.getSub());
         compVector.add(piObj);
      }
      if ((tempStr0=="b")&&(tempStr1=="g")){
         PatternInventoryObj piObj = new
         PatternInventoryObj((long)tempCoeff*ciList.getCoeff(),0,
         j*ciObj.getSub(),k*ciObj.getSub(),0);
         compVector.add(piObj);
      }
      if ((tempStr0=="b")&&(tempStr1=="y")){
         PatternInventoryObj piObj = new
         PatternInventoryObj((long)tempCoeff*ciList.getCoeff(),0,
         j*ciObj.getSub(),0,k*ciObj.getSub());
         compVector.add(piObj);
      }
      if ((tempStr0=="g")&&(tempStr1=="y")){
         PatternInventoryObj piObj = new
         PatternInventoryObj((long)tempCoeff*ciList.getCoeff(),0,
```

```
            0,j*ciObj.getSub(),k*ciObj.getSub());
            compVector.add(piObj);
        }
    }
    compVector = groupTerms(compVector);
}
//for last term in Cycle Index, expand and add
ciList = (CycleIndexList)ciVector.elementAt(ciVector.size()-1);
tempVec = ciList.getCycleIndexEntry();
CycleIndexObj tempCIObj0 = (CycleIndexObj)tempVec.elementAt(0);
Vector tempVec0 = new Vector();
CycleIndexObj tempCIObj1 = (CycleIndexObj)tempVec.elementAt(1);
Vector tempVec1 = new Vector();
for (int m=0; m<=tempCIObj0.getSup(); m++){
    int n=tempCIObj0.getSup()-m;
    float tempCoeff =
    (factorial((float)(m+n)))/((factorial((float)m))*(factorial((float)n)));
    if ((tempStr0=="r")&&(tempStr1=="b")){
        PatternInventoryObj piObj = new PatternInventoryObj((long)tempCoeff,
        m*tempCIObj0.getSub(),n*tempCIObj0.getSub(),0,0);
        tempVec0.add(piObj);
    }
    if ((tempStr0=="r")&&(tempStr1=="g")){
        PatternInventoryObj piObj = new PatternInventoryObj((long)tempCoeff,
        m*tempCIObj0.getSub(),0,n*tempCIObj0.getSub(),0);
        tempVec0.add(piObj);
    }
    if ((tempStr0=="r")&&(tempStr1=="y")){
        PatternInventoryObj piObj = new PatternInventoryObj((long)tempCoeff,
        m*tempCIObj0.getSub(),0,0,n*tempCIObj0.getSub());
        tempVec0.add(piObj);
    }
    if ((tempStr0=="b")&&(tempStr1=="g")){
        PatternInventoryObj piObj = new PatternInventoryObj((long)tempCoeff,
        0,m*tempCIObj0.getSub(),n*tempCIObj0.getSub(),0);
        tempVec0.add(piObj);
    }
    if ((tempStr0=="b")&&(tempStr1=="y")){
        PatternInventoryObj piObj = new PatternInventoryObj((long)tempCoeff,
        0,m*tempCIObj0.getSub(),0,n*tempCIObj0.getSub());
        tempVec0.add(piObj);
    }
    if ((tempStr0=="g")&&(tempStr1=="y")){
        PatternInventoryObj piObj = new PatternInventoryObj((long)tempCoeff,
        0,0,m*tempCIObj0.getSub(),n*tempCIObj0.getSub());
```

```
            tempVec0.add(piObj);
        }
    }
    tempVec0 = groupTerms(tempVec0);
    for (int p=0; p<=tempCIObj1.getSup(); p++){
        int q=tempCIObj1.getSup()-p;
        float tempCoeff =
        (factorial((float)(p+q)))/((factorial((float)p))*(factorial((float)q)));
        if ((tempStr0=="r")&&(tempStr1=="b")){
            PatternInventoryObj piObj = new PatternInventoryObj((long)tempCoeff,
            p*tempCIObj1.getSub(),q*tempCIObj1.getSub(),0,0);
            tempVec1.add(piObj);
        }
        if ((tempStr0=="r")&&(tempStr1=="g")){
            PatternInventoryObj piObj = new PatternInventoryObj((long)tempCoeff,
            p*tempCIObj1.getSub(),0,q*tempCIObj1.getSub(),0);
            tempVec1.add(piObj);
        }
        if ((tempStr0=="r")&&(tempStr1=="y")){
            PatternInventoryObj piObj = new PatternInventoryObj((long)tempCoeff,
            p*tempCIObj1.getSub(),0,0,q*tempCIObj1.getSub());
            tempVec1.add(piObj);
        }
        if ((tempStr0=="b")&&(tempStr1=="g")){
            PatternInventoryObj piObj = new PatternInventoryObj((long)tempCoeff,
            0,p*tempCIObj1.getSub(),q*tempCIObj1.getSub(),0);
            tempVec1.add(piObj);
        }
        if ((tempStr0=="b")&&(tempStr1=="y")){
            PatternInventoryObj piObj = new PatternInventoryObj((long)tempCoeff,
            0,p*tempCIObj1.getSub(),0,q*tempCIObj1.getSub());
            tempVec1.add(piObj);
        }
        if ((tempStr0=="g")&&(tempStr1=="y")){
            PatternInventoryObj piObj = new PatternInventoryObj((long)tempCoeff,
            0,0,p*tempCIObj1.getSub(),q*tempCIObj1.getSub());
            tempVec1.add(piObj);
        }
    }
    tempVec1 = groupTerms(tempVec1);
    Vector tempVec2 = multiply(tempVec0, tempVec1);
    tempVec2 = groupTerms(tempVec2);
    for (int r=0; r<tempVec2.size(); r++){
        PatternInventoryObj tempObj = (PatternInventoryObj)tempVec2.elementAt(r);
        long tempLong;
```

```
      tempLong = tempObj.getCoeff()*ciList.getCoeff();
      tempObj.setCoeff(tempLong);
      compVector.add(tempObj);
    }
    compVector = groupTerms(compVector);
    for (int s=0; s<compVector.size(); s++){
      PatternInventoryObj tempObj2 = (PatternInventoryObj)compVector.elementAt(s);
      tempObj2.setCoeff(tempObj2.getCoeff()/(2*total));
      compVector.set(s,tempObj2);
    }

}

//if three colors are used
private void computeThreeColor(){

    //create storage Vectors
    storeVec1 = new Vector(total+1);
    for (int a=0; a<total+1; a++){
      storeVec2 = new Vector(total+1);
      storeVec1.add(a, storeVec2);
      for (int c=0; c<total+1; c++){
        storeVec3 = new Vector(total+1);
        storeVec2.add(c, storeVec3);
        for (int d=0; d<total+1; d++){
          storeVec4 = new Vector(total+1);
          storeVec3.add(d, storeVec4);
          for (int e=0; e<total+1; e++){
            PatternInventoryObj tempObj = new PatternInventoryObj(0,a,c,d,e);
            storeVec4.add(e, tempObj);
          }
        }
      }
    }
    compVector = new Vector();
    String tempStr0 = (String)colors.elementAt(0);
    String tempStr1 = (String)colors.elementAt(1);
    String tempStr2 = (String)colors.elementAt(2);
    CycleIndexList ciList = new CycleIndexList();
    Vector tempVec = new Vector();
    //for each Cycle Index term but last, expand and add based on colors
    for(int i=0; i<ciVector.size()-1; i++){
      ciList = (CycleIndexList)ciVector.elementAt(i);
      tempVec = ciList.getCycleIndexEntry();
      CycleIndexObj ciObj = (CycleIndexObj)tempVec.elementAt(0);
```

```
for (int j=0; j<=ciObj.getSup(); j++){
   for (int k=ciObj.getSup()-j; k>=0; k--){
      int l=ciObj.getSup()-j-k;
      float tempCoeff = (factorial((float)(j+k+l)))/((factorial((float)j))
      *(factorial((float)k))*(factorial((float)l)));
      tempCoeff = (float)java.lang.Math.ceil((double)tempCoeff);
      if ((tempStr0=="r")&&(tempStr1=="b")&&(tempStr2=="g")){
         PatternInventoryObj piObj = new
         PatternInventoryObj((long)tempCoeff*ciList.getCoeff(),
         j*ciObj.getSub(),k*ciObj.getSub(),l*ciObj.getSub(),0);
         Vector tempVec1 = (Vector)storeVec1.elementAt(j*ciObj.getSub());
         Vector tempVec2 = (Vector)tempVec1.elementAt(k*ciObj.getSub());
         Vector tempVec3 = (Vector)tempVec2.elementAt(l*ciObj.getSub());
         PatternInventoryObj tempObj =
         (PatternInventoryObj)tempVec3.elementAt(0);
         tempObj.setCoeff(tempObj.getCoeff()
         +((long)tempCoeff*ciList.getCoeff()));
         tempVec3.set(0, tempObj);
      }
      if ((tempStr0=="r")&&(tempStr1=="b")&&(tempStr2=="y")){
         PatternInventoryObj piObj = new
         PatternInventoryObj((long)tempCoeff*ciList.getCoeff(),
         j*ciObj.getSub(),k*ciObj.getSub(),0,l*ciObj.getSub());
         Vector tempVec1 = (Vector)storeVec1.elementAt(j*ciObj.getSub());
         Vector tempVec2 = (Vector)tempVec1.elementAt(k*ciObj.getSub());
         Vector tempVec3 = (Vector)tempVec2.elementAt(0);
         PatternInventoryObj tempObj = (PatternInventoryObj)
         tempVec3.elementAt(l*ciObj.getSub());
         tempObj.setCoeff(tempObj.getCoeff()
         +((long)tempCoeff*ciList.getCoeff()));
         tempVec3.set(l*ciObj.getSub(), tempObj);
      }
      if ((tempStr0=="r")&&(tempStr1=="g")&&(tempStr2=="y")){
         PatternInventoryObj piObj = new
         PatternInventoryObj((long)tempCoeff*ciList.getCoeff(),
         j*ciObj.getSub(),0,k*ciObj.getSub(),l*ciObj.getSub());
         Vector tempVec1 = (Vector)storeVec1.elementAt(j*ciObj.getSub());
         Vector tempVec2 = (Vector)tempVec1.elementAt(0);
         Vector tempVec3 = (Vector)tempVec2.elementAt(k*ciObj.getSub());
         PatternInventoryObj tempObj = (PatternInventoryObj)
         tempVec3.elementAt(l*ciObj.getSub());
         tempObj.setCoeff(tempObj.getCoeff()
         +((long)tempCoeff*ciList.getCoeff()));
         tempVec3.set(l*ciObj.getSub(), tempObj);
      }
```

```
        if ((tempStr0=="b")&&(tempStr1=="g")&&(tempStr2=="y")){
          PatternInventoryObj piObj = new
          PatternInventoryObj((long)tempCoeff*ciList.getCoeff(),
          0,j*ciObj.getSub(),k*ciObj.getSub(),l*ciObj.getSub());
          Vector tempVec1 = (Vector)storeVec1.elementAt(0);
          Vector tempVec2 = (Vector)tempVec1.elementAt(j*ciObj.getSub());
          Vector tempVec3 = (Vector)tempVec2.elementAt(k*ciObj.getSub());
          PatternInventoryObj tempObj = (PatternInventoryObj)
          tempVec3.elementAt(l*ciObj.getSub());
          tempObj.setCoeff(tempObj.getCoeff()
          +((long)tempCoeff*ciList.getCoeff()));
          tempVec3.set(l*ciObj.getSub(), tempObj);
        }
      }
    }
  }
  //for last term in Cycle Index, expand and add
  ciList = (CycleIndexList)ciVector.elementAt(ciVector.size()-1);
  tempVec = ciList.getCycleIndexEntry();
  CycleIndexObj tempCIObj0 = (CycleIndexObj)tempVec.elementAt(0);
  Vector tempVec0 = new Vector();
  CycleIndexObj tempCIObj1 = (CycleIndexObj)tempVec.elementAt(1);
  Vector tempVec1 = new Vector();
  for (int m=0; m<=tempCIObj0.getSup(); m++){
    for (int n=tempCIObj0.getSup()-m; n>=0; n--){
      int o=tempCIObj0.getSup()-m-n;
      float tempCoeff = (factorial((float)(m+n+o)))/((factorial((float)m))
      *(factorial((float)n))*(factorial((float)o)));
      tempCoeff = (float)java.lang.Math.ceil((double)tempCoeff);
      if ((tempStr0=="r")&&(tempStr1=="b")&&(tempStr2=="g")){
        PatternInventoryObj piObj = new PatternInventoryObj((long)tempCoeff,
        m*tempCIObj0.getSub(),n*tempCIObj0.getSub(),
        o*tempCIObj0.getSub(),0);
        tempVec0.add(piObj);
      }
      if ((tempStr0=="r")&&(tempStr1=="b")&&(tempStr2=="y")){
        PatternInventoryObj piObj = new PatternInventoryObj((long)tempCoeff,
        m*tempCIObj0.getSub(),n*tempCIObj0.getSub(),0,
        o*tempCIObj0.getSub());
        tempVec0.add(piObj);
      }
      if ((tempStr0=="r")&&(tempStr1=="g")&&(tempStr2=="y")){
        PatternInventoryObj piObj = new PatternInventoryObj((long)tempCoeff,
        m*tempCIObj0.getSub(),0,n*tempCIObj0.getSub(),
        o*tempCIObj0.getSub());
```

```
            tempVec0.add(piObj);
          }
          if ((tempStr0=="b")&&(tempStr1=="g")&&(tempStr2=="y")){
            PatternInventoryObj piObj = new PatternInventoryObj((long)tempCoeff,
            0,m*tempCIObj0.getSub(),n*tempCIObj0.getSub(),
            o*tempCIObj0.getSub());
            tempVec0.add(piObj);
          }
        }
      }
      tempVec0 = groupTerms(tempVec0);
      for (int p=0; p<=tempCIObj1.getSup(); p++){
        for (int q=tempCIObj1.getSup()-p; q>=0; q--){
          int r=tempCIObj1.getSup()-p-q;
          float tempCoeff = (factorial((float)(p+q+r)))/((factorial((float)p))
          *(factorial((float)q))*(factorial((float)r)));
          tempCoeff = (float)java.lang.Math.ceil((double)tempCoeff);
          if ((tempStr0=="r")&&(tempStr1=="b")&&(tempStr2=="g")){
            for (int a=0; a<tempVec0.size(); a++){
              PatternInventoryObj piObj = new PatternInventoryObj((long)tempCoeff,

p*tempCIObj1.getSub(),q*tempCIObj1.getSub(),r*tempCIObj1.getSub(),0);
              PatternInventoryObj piObj2 =
              (PatternInventoryObj)tempVec0.elementAt(a);
              piObj.setCoeff(piObj.getCoeff()*piObj2.getCoeff()*ciList.getCoeff());
              piObj.setRExp(piObj.getRExp()+piObj2.getRExp());
              piObj.setBExp(piObj.getBExp()+piObj2.getBExp());
              piObj.setGExp(piObj.getGExp()+piObj2.getGExp());
              piObj.setYExp(piObj.getYExp()+piObj2.getYExp());
              tempVec1 = (Vector)storeVec1.elementAt(piObj.getRExp());
              Vector tempVec2 = (Vector)tempVec1.elementAt(piObj.getBExp());
              Vector tempVec3 = (Vector)tempVec2.elementAt(piObj.getGExp());
              PatternInventoryObj tempObj = (PatternInventoryObj)
              tempVec3.elementAt(piObj.getYExp());
              tempObj.setCoeff(tempObj.getCoeff()+piObj.getCoeff());
              tempVec3.set(tempObj.getYExp(), tempObj);
            }
          }
          if ((tempStr0=="r")&&(tempStr1=="b")&&(tempStr2=="y")){
            for (int a=0; a<tempVec0.size(); a++){
              PatternInventoryObj piObj = new PatternInventoryObj((long)tempCoeff,
              p*tempCIObj1.getSub(),q*tempCIObj1.getSub(),
              0,r*tempCIObj1.getSub());
              PatternInventoryObj piObj2 =
              (PatternInventoryObj)tempVec0.elementAt(a);
```

```
        piObj.setCoeff(piObj.getCoeff()*piObj2.getCoeff()*ciList.getCoeff());
        piObj.setRExp(piObj.getRExp()+piObj2.getRExp());
        piObj.setBExp(piObj.getBExp()+piObj2.getBExp());
        piObj.setGExp(piObj.getGExp()+piObj2.getGExp());
        piObj.setYExp(piObj.getYExp()+piObj2.getYExp());
        tempVec1 = (Vector)storeVec1.elementAt(piObj.getRExp());
        Vector tempVec2 = (Vector)tempVec1.elementAt(piObj.getBExp());
        Vector tempVec3 = (Vector)tempVec2.elementAt(piObj.getGExp());
        PatternInventoryObj tempObj = (PatternInventoryObj)
        tempVec3.elementAt(piObj.getYExp());
        tempObj.setCoeff(tempObj.getCoeff()+piObj.getCoeff());
        tempVec3.set(tempObj.getYExp(), tempObj);
    }
}
if ((tempStr0=="r")&&(tempStr1=="g")&&(tempStr2=="y")){
    for (int a=0; a<tempVec0.size(); a++){
        PatternInventoryObj piObj = new PatternInventoryObj((long)tempCoeff,
        p*tempCIObj1.getSub(),0,q*tempCIObj1.getSub(),
        r*tempCIObj1.getSub());
        PatternInventoryObj piObj2 =
        (PatternInventoryObj)tempVec0.elementAt(a);
        piObj.setCoeff(piObj.getCoeff()*piObj2.getCoeff()*ciList.getCoeff());
        piObj.setRExp(piObj.getRExp()+piObj2.getRExp());
        piObj.setBExp(piObj.getBExp()+piObj2.getBExp());
        piObj.setGExp(piObj.getGExp()+piObj2.getGExp());
        piObj.setYExp(piObj.getYExp()+piObj2.getYExp());
        tempVec1 = (Vector)storeVec1.elementAt(piObj.getRExp());
        Vector tempVec2 = (Vector)tempVec1.elementAt(piObj.getBExp());
        Vector tempVec3 = (Vector)tempVec2.elementAt(piObj.getGExp());
        PatternInventoryObj tempObj = (PatternInventoryObj)
        tempVec3.elementAt(piObj.getYExp());
        tempObj.setCoeff(tempObj.getCoeff()+piObj.getCoeff());
        tempVec3.set(tempObj.getYExp(), tempObj);
    }
}
if ((tempStr0=="b")&&(tempStr1=="g")&&(tempStr2=="y")){
    for (int a=0; a<tempVec0.size(); a++){
        PatternInventoryObj piObj = new PatternInventoryObj((long)tempCoeff,
        0,p*tempCIObj1.getSub(),q*tempCIObj1.getSub(),
        r*tempCIObj1.getSub());
        PatternInventoryObj piObj2 =
        (PatternInventoryObj)tempVec0.elementAt(a);
        piObj.setCoeff(piObj.getCoeff()*piObj2.getCoeff()*ciList.getCoeff());
        piObj.setRExp(piObj.getRExp()+piObj2.getRExp());
        piObj.setBExp(piObj.getBExp()+piObj2.getBExp());
```

```
                piObj.setGExp(piObj.getGExp()+piObj2.getGExp());
                piObj.setYExp(piObj.getYExp()+piObj2.getYExp());
                tempVec1 = (Vector)storeVec1.elementAt(piObj.getRExp());
                Vector tempVec2 = (Vector)tempVec1.elementAt(piObj.getBExp());
                Vector tempVec3 = (Vector)tempVec2.elementAt(piObj.getGExp());
                PatternInventoryObj tempObj = (PatternInventoryObj)
                tempVec3.elementAt(piObj.getYExp());
                tempObj.setCoeff(tempObj.getCoeff()+piObj.getCoeff());
                tempVec3.set(tempObj.getYExp(), tempObj);
            }
        }
    }
  }
  for (int u=storeVec1.size()-1; u>=0; u--){
    Vector storeVec2 = (Vector)storeVec1.elementAt(u);
    for (int v=storeVec2.size()-1 ; v>=0; v--){
      Vector storeVec3 = (Vector)storeVec2.elementAt(v);
      for (int w=storeVec3.size()-1; w>=0; w--){
        Vector storeVec4 = (Vector)storeVec3.elementAt(w);
        for (int z=storeVec4.size()-1; z>=0; z--){
          PatternInventoryObj tempObj =
          (PatternInventoryObj)storeVec4.elementAt(z);
          if (tempObj.getCoeff()!=0){
            tempObj.setCoeff(tempObj.getCoeff()/(2*total));
            compVector.add(tempObj);
          }
        }
      }
    }
  }

}

//if four colors used
private void computeFourColor(){

  //create storage
  storeVec1 = new Vector(total+1);
  for (int a=0; a<total+1; a++){
    storeVec2 = new Vector(total+1);
    storeVec1.add(a, storeVec2);
    for (int c=0; c<total+1; c++){
      storeVec3 = new Vector(total+1);
      storeVec2.add(c, storeVec3);
      for (int d=0; d<total+1; d++){
```

```
            storeVec4 = new Vector(total+1);
            storeVec3.add(d, storeVec4);
            for (int e=0; e<total+1; e++){
               PatternInventoryObj tempObj = new PatternInventoryObj(0,a,c,d,e);
               storeVec4.add(e, tempObj);
            }
         }
      }
   }
   //for each Cycle Index term but last, expand and add
   compVector = new Vector();
   CycleIndexList ciList = new CycleIndexList();
   Vector tempVec = new Vector();
   for(int i=0; i<ciVector.size()-1; i++){
      ciList = (CycleIndexList)ciVector.elementAt(i);
      tempVec = ciList.getCycleIndexEntry();
      CycleIndexObj ciObj = (CycleIndexObj)tempVec.elementAt(0);
      for (int j=0; j<=ciObj.getSup(); j++){
         for (int k=ciObj.getSup()-j; k>=0; k--){
            for (int l=ciObj.getSup()-j-k; l>=0; l--){
               int m=ciObj.getSup()-j-k-l;
               float tempCoeff = (factorial((float)(j+k+l+m)))/((factorial((float)j))
               *(factorial((float)k))*(factorial((float)l))*(factorial((float)m)));
               tempCoeff = (float)java.lang.Math.ceil((double)tempCoeff);
               PatternInventoryObj piObj = new PatternInventoryObj
               ((long)tempCoeff*ciList.getCoeff(),
               j*ciObj.getSub(),k*ciObj.getSub(),l*ciObj.getSub(),m*ciObj.getSub());
               Vector tempVec1 = (Vector)storeVec1.elementAt(j*ciObj.getSub());
               Vector tempVec2 = (Vector)tempVec1.elementAt(k*ciObj.getSub());
               Vector tempVec3 = (Vector)tempVec2.elementAt(l*ciObj.getSub());
               PatternInventoryObj tempObj = (PatternInventoryObj)
               tempVec3.elementAt(m*ciObj.getSub());
               tempObj.setCoefff(tempObj.getCoeff()
               +((long)tempCoeff*ciList.getCoeff()));
               tempVec3.set(m*ciObj.getSub(), tempObj);
            }
         }
      }
   }
   //for last term in Cycle Index, expand and add
   ciList = (CycleIndexList)ciVector.elementAt(ciVector.size()-1);
   tempVec = ciList.getCycleIndexEntry();
   CycleIndexObj tempCIObj0 = (CycleIndexObj)tempVec.elementAt(0);
   Vector tempVec0 = new Vector();
   CycleIndexObj tempCIObj1 = (CycleIndexObj)tempVec.elementAt(1);
```

```
Vector tempVec1 = new Vector();
for (int n=0; n<=tempCIObj0.getSup(); n++){
   for (int o=tempCIObj0.getSup()-n; o>=0; o--){
      for (int p=tempCIObj0.getSup()-n-o; p>=0; p--){
         int q=tempCIObj0.getSup()-n-o-p;
         float tempCoeff = (factorial((float)(n+o+p+q)))/((factorial((float)n))
         *(factorial((float)o))*(factorial((float)p))*(factorial((float)q)));
         tempCoeff = (float)java.lang.Math.ceil((double)tempCoeff);
         PatternInventoryObj piObj = new PatternInventoryObj
         ((long)tempCoeff,n*tempCIObj0.getSub(),o*tempCIObj0
         .getSub(),p*tempCIObj0.getSub(),q*tempCIObj0.getSub());
         tempVec0.add(piObj);
      }
   }
}
tempVec0 = groupTerms(tempVec0);
for (int r=0; r<=tempCIObj1.getSup(); r++){
   for (int s=tempCIObj1.getSup()-r; s>=0; s--){
      for (int t=tempCIObj1.getSup()-r-s; t>=0; t--){
         int u=tempCIObj1.getSup()-r-s-t;
         float tempCoeff = (factorial((float)(r+s+t+u)))/((factorial((float)r))
         *(factorial((float)s))*(factorial((float)t))*(factorial((float)u)));
         tempCoeff = (float)java.lang.Math.ceil((double)tempCoeff);
         for (int a=0; a<tempVec0.size(); a++){
            PatternInventoryObj piObj = new
            PatternInventoryObj((long)tempCoeff,r*tempCIObj1
            .getSub(),s*tempCIObj1.getSub(),t*tempCIObj1.getSub(),
            u*tempCIObj1.getSub());
            PatternInventoryObj piObj2 =
            (PatternInventoryObj)tempVec0.elementAt(a);
            piObj.setCoeff(piObj.getCoeff()*piObj2.getCoeff()*ciList.getCoeff());
            piObj.setRExp(piObj.getRExp()+piObj2.getRExp());
            piObj.setBExp(piObj.getBExp()+piObj2.getBExp());
            piObj.setGExp(piObj.getGExp()+piObj2.getGExp());
            piObj.setYExp(piObj.getYExp()+piObj2.getYExp());
            tempVec1 = (Vector)storeVec1.elementAt(piObj.getRExp());
            Vector tempVec2 = (Vector)tempVec1.elementAt(piObj.getBExp());
            Vector tempVec3 = (Vector)tempVec2.elementAt(piObj.getGExp());
            PatternInventoryObj tempObj = (PatternInventoryObj)
            tempVec3.elementAt(piObj.getYExp());
            tempObj.setCoeff(tempObj.getCoeff()+piObj.getCoeff());
            tempVec3.set(tempObj.getYExp(), tempObj);
         }
      }
   }
```

```
        }

        for (int u=storeVec1.size()-1; u>=0; u--){
           Vector storeVec2 = (Vector)storeVec1.elementAt(u);
           for (int v=storeVec2.size()-1 ; v>=0; v--){
              Vector storeVec3 = (Vector)storeVec2.elementAt(v);
              for (int w=storeVec3.size()-1; w>=0; w--){
                 Vector storeVec4 = (Vector)storeVec3.elementAt(w);
                 for (int z=storeVec4.size()-1; z>=0; z--){
                    PatternInventoryObj tempObj =
                    (PatternInventoryObj)storeVec4.elementAt(z);
                    if (tempObj.getCoeff()!=0){
                       tempObj.setCoeff(tempObj.getCoeff()/(2*total));
                          compVector.add(tempObj);
                    }
                 }
              }
           }
        }

     }


}
```

```
/*
 * PatternInventoryObj.java
 * For use with the Necklace Illustrator Applet
 * by David Fraser
 * Rochester Institute of Technology
 * May 2005
 *
 */

package Necklace;

public class PatternInventoryObj {

   long coeff = 1;
   int rExp = 0;
   int bExp = 0;
   int gExp = 0;
   int yExp = 0;

   //create a new PatternInventoryObj
   public PatternInventoryObj() {
   }

   //create a new PatternInventoryObj that is passed coefficient and color exponents
   public PatternInventoryObj(long c, int r, int b, int g, int y){

      coeff=c;
      rExp=r;
      bExp=b;
      gExp=g;
      yExp=y;

   }

   //set the coefficient
   public void setCoeff(long c){
      coeff=c;
   }

   //set the red exponent
   public void setRExp(int r){
      rExp=r;
   }

   //set the blue exponent
```

```
    public void setBExp(int b){
        bExp=b;
    }

    //set the green exponent
    public void setGExp(int g){
        gExp=g;
    }

    //set the yellow exponent
    public void setYExp(int y){
        yExp=y;
    }

    //get the coefficient
    public long getCoeff(){
        return coeff;
    }

    //get the red exponent
    public int getRExp(){
        return rExp;
    }

    //get the blue exponent
    public int getBExp(){
        return bExp;
    }

    //get the green exponent
    public int getGExp(){
        return gExp;
    }

    //get the yellow exponent
    public int getYExp(){
        return yExp;
    }

}
```

```
/*
 * PIDraw.java
 * For use with the Necklace Illustrator Applet
 * by David Fraser
 * Rochester Institute of Technology
 * May 2005
 *
 */

package Necklace;

import java.awt.Canvas;
import java.awt.Component;
import java.util.Vector;
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Font;
import java.lang.Integer;
import Necklace.PatternInventoryObj;
import java.awt.FontMetrics;
import java.awt.image.BufferedImage;
import java.awt.image.ImageObserver;

public class PIDraw extends Canvas implements ImageObserver{

   Vector piVector;
   Graphics g;
   BufferedImage img;
   int newY;
   int red;
   int blue;
   int green;
   int yellow;
   int maxNecklace;

   //create a new PIDraw that is passed the Pattern Inventory Vector and color integers
   public PIDraw(Vector vec, int r, int b, int gre, int yel) {

      piVector = vec;
      red = r;
      blue = b;
      green = gre;
      yellow = yel;
      maxNecklace=0;
      img = new BufferedImage(1,1,2);
```

```
    makeImage();

}

//paint method to draw the Pattern Inventory
public void paint(Graphics g){

    g.drawImage(img,0,0,Color.WHITE,this);
    g.dispose();

}

private void makeImage(){

    //make a new Buffered Image to draw Pattern Inventory
    img = new BufferedImage(650, 7000, 2);
    Graphics g = img.createGraphics();
    Font boldFont = new Font("boldFont", java.awt.Font.BOLD, 12);
    g.setFont(boldFont);
    g.setColor(Color.BLACK);
    g.drawString("Pattern Inventory:  ",  10, 20);
    int newX = 10+g.getFontMetrics(boldFont).stringWidth("PatternInventory:  ");
    newY = 20;
    Integer convInt = new Integer(0);
    //if Pattern Inventory Vector is not empty
    if (piVector.size()!=0){
        Font regFont = new Font("regFont", java.awt.Font.PLAIN, 12);
        Font smallFont = new Font("smallFont", java.awt.Font.PLAIN, 10);
        Font smallBold = new Font("smallBold", java.awt.Font.BOLD, 10);
        String tempStr;
        //for each term in Pattern Inventory
        for (int i=0; i<piVector.size(); i++){
            PatternInventoryObj pattObj = (PatternInventoryObj)piVector.elementAt(i);
            //if exponents match color seletion, draw in bold
            if ((pattObj.getRExp()==red)&&(pattObj.getBExp()==blue)&&
            (pattObj.getGExp()==green)&&(pattObj.getYExp()==yellow)){
                maxNecklace = (int)pattObj.getCoeff();
                //if coefficient is not 0, draw it
                if (pattObj.getCoeff()!=1){
                    tempStr = convInt.toString((int)pattObj.getCoeff());
                    g.setFont(boldFont);
                    g.drawString(tempStr, newX, newY);
                    newX = newX+g.getFontMetrics(boldFont).stringWidth(tempStr);
                }
                //if red exponent is not 0, draw it
```

```
    if (pattObj.getRExp()!=0){
       tempStr = convInt.toString(pattObj.getRExp());
       g.setFont(boldFont);
       g.drawString("r", newX, newY);
       newX = newX+g.getFontMetrics(boldFont).stringWidth("r");
       g.setFont(smallBold);
       g.drawString(tempStr, newX, (newY-4));
       newX = newX+g.getFontMetrics(smallBold).stringWidth(tempStr);
    }
    //if blue exponent is not 0, draw it
    if (pattObj.getBExp()!=0){
       tempStr = convInt.toString(pattObj.getBExp());
       g.setFont(boldFont);
       g.drawString("b", newX, newY);
       newX = newX+g.getFontMetrics(boldFont).stringWidth("b");
       g.setFont(smallBold);
       g.drawString(tempStr, newX, (newY-4));
       newX = newX+g.getFontMetrics(smallBold).stringWidth(tempStr);
    }
    //if green exponent is not 0, draw it
    if (pattObj.getGExp()!=0){
       tempStr = convInt.toString(pattObj.getGExp());
       g.setFont(boldFont);
       g.drawString("g", newX, newY);
       newX = newX+g.getFontMetrics(boldFont).stringWidth("g");
       g.setFont(smallBold);
       g.drawString(tempStr, newX, (newY-4));
       newX = newX+g.getFontMetrics(smallBold).stringWidth(tempStr);
    }
    //if yellow exponent is not 0, draw it
    if (pattObj.getYExp()!=0){
       tempStr = convInt.toString(pattObj.getYExp());
       g.setFont(boldFont);
       g.drawString("y", newX, newY);
       newX = newX+g.getFontMetrics(boldFont).stringWidth("y");
       g.setFont(smallBold);
       g.drawString(tempStr, newX, (newY-4));
       newX = newX+g.getFontMetrics(smallBold).stringWidth(tempStr);

    }
  //if exponents do not match color selection
  } else {
    //if coefficient is not 0, draw it
    if (pattObj.getCoeff()!=1){
       tempStr = convInt.toString((int)pattObj.getCoeff());
```

```
            g.setFont(regFont);
            g.drawString(tempStr, newX, newY);
            newX = newX+g.getFontMetrics(regFont).stringWidth(tempStr);
        }
        //if red exponent is not 0, draw it
        if (pattObj.getRExp()!=0){
            tempStr = convInt.toString(pattObj.getRExp());
            g.setFont(regFont);
            g.drawString("r", newX, newY);
            newX = newX+g.getFontMetrics(regFont).stringWidth("r");
            g.setFont(smallFont);
            g.drawString(tempStr, newX, (newY-4));
            newX = newX+g.getFontMetrics(smallFont).stringWidth(tempStr);
        }
        //if blue exponent is not 0, draw it
        if (pattObj.getBExp()!=0){
            tempStr = convInt.toString(pattObj.getBExp());
            g.setFont(regFont);
            g.drawString("b", newX, newY);
            newX = newX+g.getFontMetrics(regFont).stringWidth("b");
            g.setFont(smallFont);
            g.drawString(tempStr, newX, (newY-4));
            newX = newX+g.getFontMetrics(smallFont).stringWidth(tempStr);
        }
        //if green exponent is not 0, draw it
        if (pattObj.getGExp()!=0){
            tempStr = convInt.toString(pattObj.getGExp());
            g.setFont(regFont);
            g.drawString("g", newX, newY);
            newX = newX+g.getFontMetrics(regFont).stringWidth("g");
            g.setFont(smallFont);
            g.drawString(tempStr, newX, (newY-4));
            newX = newX+g.getFontMetrics(smallFont).stringWidth(tempStr);
        }
        //if yellow exponent is not 0, draw it
        if (pattObj.getYExp()!=0){
            tempStr = convInt.toString(pattObj.getYExp());
            g.setFont(regFont);
            g.drawString("y", newX, newY);
            newX = newX+g.getFontMetrics(regFont).stringWidth("y");
            g.setFont(smallFont);
            g.drawString(tempStr, newX, (newY-4));
            newX = newX+g.getFontMetrics(smallFont).stringWidth(tempStr);
        }
    }
```

```
            //draw " + " between terms
            if (i!=piVector.size()-1){
               g.setFont(regFont);
               g.drawString(" + ", newX, newY);
               newX = newX+g.getFontMetrics(regFont).stringWidth(" + ");
               if (newX>500){
                  newX = 10;
                  newY = newY + 20;

               }
            }
         }
      }

   }


   //return image height
   public int getHeight(){
      return newY;
   }

   //return maximum number of unique necklaces
   public int getMax(){
      return maxNecklace;
   }

}
```