

DOCUMENTATION

OVERVIEW

The basic overview of the project is to display the numerous types of representations available for a permutation.

First, user will either generate or enter a permutation. Next, he'll be able to see the one-line and the cyclic notations of that permutation.

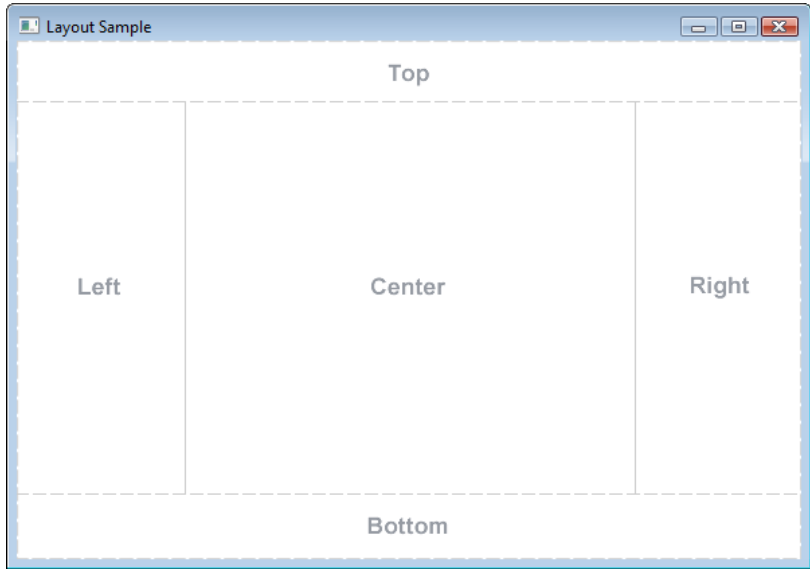
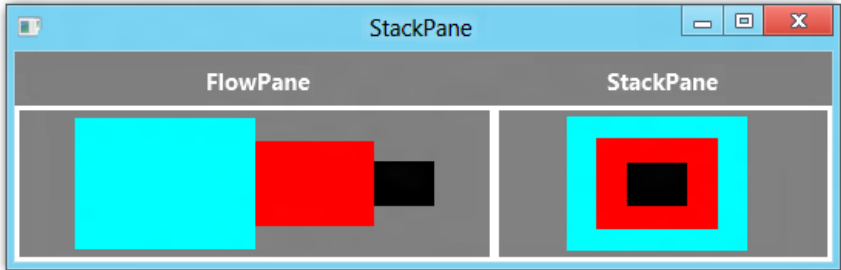
After that, he'll be able to access the four partitions and insert representations of his choice inside of them.

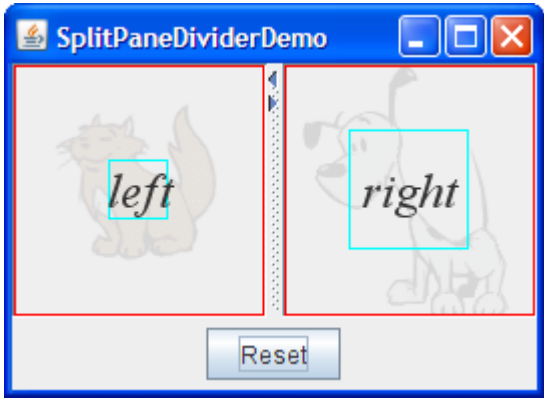
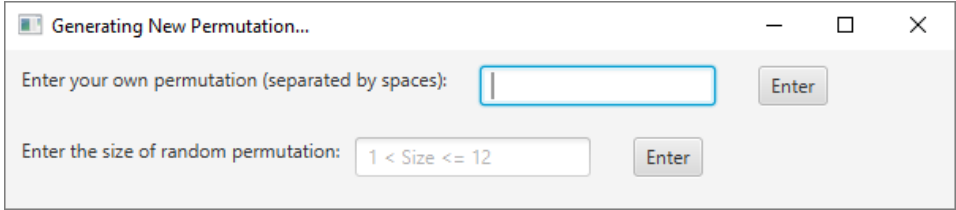

At any time, he can access help, or he can generate a new permutation. If he chooses to generate, then all of the partitions of the representations from the previous permutation will be immediately cleared.

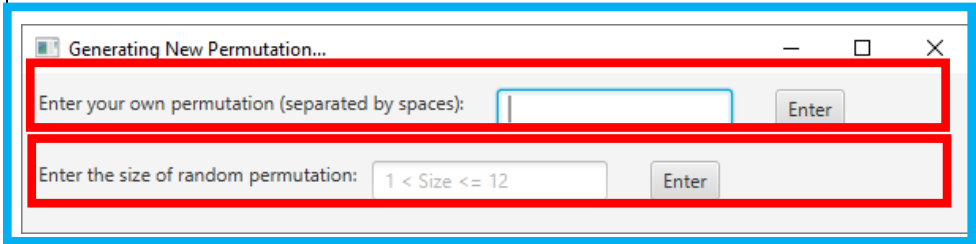
TERMINOLOGY

I've tried my best to document the terminology of JavaFX that I will be using throughout this entire project. Some of the listed terms below are not exactly reserved keywords of JavaFX toolkit. Some are simply included for the sake of simplifying vocabulary terms.

BASIC DEFINITIONS	
Stage	Running a JavaFX application means that a new window will be opened/launched. This window is referred to as the <i>Stage</i> in this toolkit.
Scene	The 'content/stuff inside of the window' is referred to as the <i>Scene</i> .
Children	This isn't exactly a keyword in JavaFX, but rather it's a simple vocabulary term. A child, in this case, is similarly defined like how a child would be defined for a graph. In this case, the <i>Scene</i> is the graph and all the contents/features/boxes/buttons/etc. inside it are the <i>children</i> of the <i>Scene</i> .
Pane	A layout which is especially helpful when we want the position of children to be very specific.
BorderPane	It's one of the layouts available in <i>Stage</i> .

	 <p>Image credits:</p> <p>https://docs.oracle.com/javase/8/javafx/api/javafx/scene/layout/BorderPane.html</p>
StackPane	<p>Another type of layout in JavaFX that lays its children in a back-to-front format in a stack data structure.</p>  <p>A FlowPane is not used in this project, so it's not defined in another row of this table. But a FlowPane, like its name, organizes its children in a horizontal or vertical manner. (Thinking of a flow chart might help.)</p> <p>Image credits: https://hajsoftutorial.com/javafx-tutorial/javafx-stackpane/</p>
SplitPane	<p>Used for dividing the scene into five parts. (For this project, there are four equal partitions and one center partition)</p>

	 <p>Image credits: https://docs.oracle.com/javase/tutorial/uiswing/components/splitpane.html</p>
Color	<p>This isn't anything major, but it was very much used to define colors for nodes.</p> <pre>setFill(Color.RED); //example</pre>
Label	 <p>In the above picture, the non-editable text to the left of the white boxes are known as labels.</p>
TextField	<p>In the above picture, the white box is the text field which allows the user to enter text.</p>
Text	<p>This also isn't related to data structures but used throughout the program to define text inside of buttons, to display errors, etc.</p>
Button	<p>A button that will perform an action upon mouse click (or any other operation that you define).</p> 

HBox VBox	<p>Two types of layouts that store objects in a straight, horizontal manner and vertical manner respectively.</p>  <p>The red boxes represent fields that are the <i>children</i> of an HBox.</p> <p>The blue box contains the two HBox's and these two are the children of a VBox.</p>
MouseEvent <i>e</i> (or <i>event</i>)	<p>The mouse events are represented by <i>e</i> or <i>event</i>. Sometimes, the program will read a mouse event <i>inside</i> another mouse event handler.</p> <p>In this case, a new identifier <i>cannot</i> be used until and unless the mouse event variable is declared static. Since mouse events are extremely sensitive and important to the program, these variables are not declared as static and the default identifier <i>e</i> or <i>event</i> is used. (If declared static for every mouse event, there will be about 30-40 new variables that will be difficult to keep track of.)</p> <p>Since this is a very large project with numerous classes, files, and methods, using as less heap memory as possible was the priority.</p>
Exception <i>e</i> (or <i>e1</i>)	<p>While this is not exactly a feature available solely in JavaFX, exception handling was used all throughout the program to provide a consistency.</p> <p>Exception handling is available in Java and for this project, it is used in both graphical and internal representations.</p> <p>The variable <i>e</i> is a common one when defining exceptions.</p>
Path	<p>Has the ability to generate a geometric shape in the form of line, curve, arc, etc.</p>
MoveTo	<p>The path is moved to a specific place using this method.</p>
QuadCurveTo	<p>Draws a quadratic Bezier curve, defined by 3 points: two endpoints and a vertex.</p>

FILES


Unlike the previous projects where all the classes were together in one single file, the major project was broken up into different files.

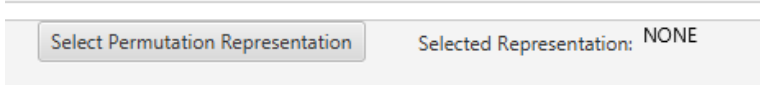
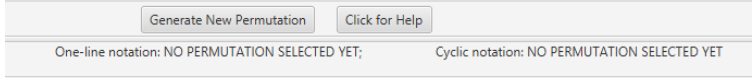
FILES PRESENT IN SHORT PROJECT #3	
<i>MainProgram.java</i> (Class)	The main class where the program is run from and executed.
<i>GeneratePermutation.java</i> (Class)	Open a new stage and ask the user to input either a permutation or the length for a random permutation generation.
<i>SquareGrid.java</i> (Class)	Generate a square grid that has colored squares (marked by red) and inversions (marked by green) in the appropriate pane.
<i>Inverse.java</i> (Class)	Generate the permutation's inverse in the appropriate pane.
<i>InversionTable</i> (Class)	Generate the permutation's inversion table in the appropriate pane.
<i>FactoradicNumber</i> (Class)	Generate the permutation's factoradic number in the appropriate pane.
<i>GraphThePermutation.java</i> (Class)	Generate the permutation graph in the appropriate pane.
<i>Meanders.java</i> (Class)	Generate the permutation's meander in the appropriate pane.
<i>GridTours</i> (Class)	Generate the permutation's grid tour representation in the appropriate pane.

UNDERSTANDING IDENTIFIERS

The project is divided into multiple files. Therefore, the identifiers are also listed according to the usage of each file. However, any unimportant/non-crucial identifiers such as the local ones declared in loops, if-else statements, and other dummy variables are not listed. Only those that might help/aid in understanding the program better are documented carefully.

Also, the majority of the variables in the following tables are static variables because they're accessed among each file. In order for a file to access another file's variable, both these files must have that variable in heap memory. The way to do that is to make that particular variable as a static one. And thus, the variable will be made public for use.

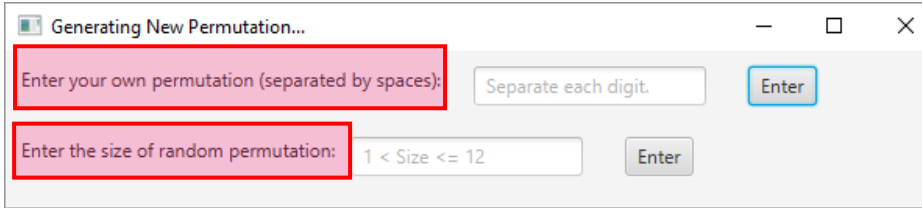
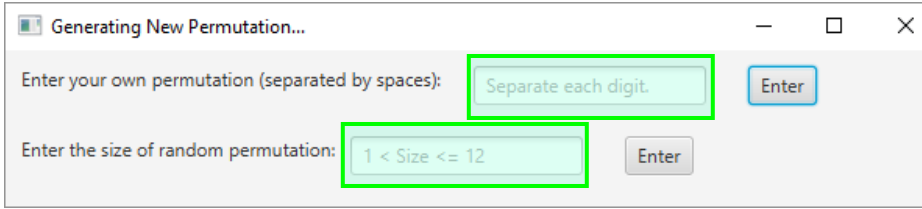
<p style="text-align: center;">MAIN CLASS: <i>MainProgram.java</i> (Class)</p> <p style="text-align: center;">(GRAPHICS PORTION)</p>	
<i>rootPane</i> (<i>BorderPane</i>)	Used as a basic, underlying layout in the scene to organize visual content in the screen.
<i>canvas1</i> (<i>SplitPane</i>) <i>canvas2</i> (<i>SplitPane</i>) <i>canvas3</i> (<i>SplitPane</i>) <i>canvas4</i> (<i>SplitPane</i>)	Each ‘canvas’ represents each of the four partitions present in the screen.
<i>forPermutationEntry</i> (<i>SplitPane</i>)	This represents the center partition of the screen which allows the user to generate a permutation and view that result as well.
<i>generatePermutation</i> (<i>Button</i>)	The button inside of the above pane that opens up a new stage and asks the user to either input his own permutation or input a length to generate a random permutation.
<i>help</i> (<i>Button</i>)	Access help at any time, regarding the different permutation representations.
<i>displayOneLinePermutation</i> (<i>Label</i>)	This is used to denote the permutation that user generated.
<i>displayCyclicNotation</i> (<i>Label</i>)	This is used to representation the permutation in a cyclic notation.
<i>menu1</i> (<i>Button</i>) <i>menu2</i> (<i>Button</i>) <i>menu3</i> (<i>Button</i>) <i>menu4</i> (<i>Button</i>)	<p>There are four partitions in the screen. Each partition will have a separate menu that will enable the user to access permutation representations.</p> <p><i>There are seven in total.</i></p>
<i>label1</i> (<i>Label</i>) <i>label2</i> (<i>Label</i>) <i>label3</i> (<i>Label</i>) <i>label4</i> (<i>Label</i>)	<p>Next, a separate label is used to tell the user <i>what</i> representation is being displayed.</p> <p>After that, the selected representation is worded and displayed after the label.</p>
<i>representation1</i> (<i>Text</i>) <i>representation2</i> (<i>Text</i>)	

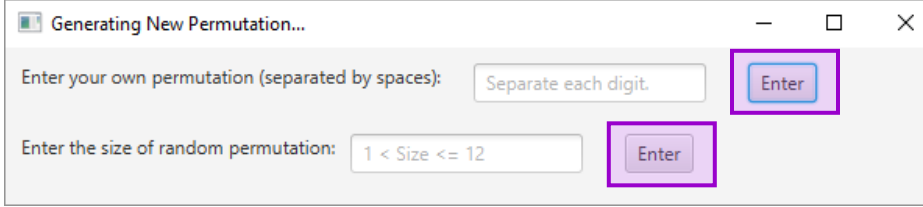
<i>representation3</i> (Text) <i>representation4</i> (Text)	<p>The red box represents the button to access menu. The green box represents the unmodifiable label. The purple box represents the selected representation.</p>
<i>selectedRepresentation1</i> (Text) <i>selectedRepresentation2</i> (Text) <i>selectedRepresentation3</i> (Text) <i>selectedRepresentation4</i> (Text)	<p>The representation from above is just stored in a separate variable.</p>
<i>canvasBox1</i> (HBox) <i>canvasBox2</i> (HBox) <i>canvasBox3</i> (HBox) <i>canvasBox4</i> (HBox)	<p>Each of the four partitions will contain the following. The following three things are displayed in a horizontal manner using a corresponding <i>HBox</i>.</p> 
<i>centerBox</i> (VBox)	<p>The following is represented using a <i>VBox</i>. Generating permutation and clicking for help buttons are represented with the help of a <i>HBox</i>, but this HBox is defined as a local variable.</p> 
<i>permutation</i> (ArrayList)	<p>The entered or randomly generated permutation is stored in an <i>ArrayList</i>.</p>
<i>grid</i> (SquareGrid)	<p>An object of the square grid representation, implemented in <i>SquareGrid.java</i>.</p>
<i>inversePermutation</i> (Inverse)	<p>An object of the inverse representation, implemented in <i>Inverse.java</i>.</p>
<i>inversion</i> (InversionTable)	<p>An object of the inversion table representation, implemented in <i>InversionTable.java</i>.</p>
<i>factoradicNumber</i> (FactoradicNumber)	<p>An object of the factoradic number representation, implemented in <i>FactoradicNumber.java</i>.</p>

<i>drawGraph</i> (<i>GraphThePermutation</i>)	An object of the permutation graph representation, implemented in <i>GraphThePermutation.java</i> .
<i>meanders</i> (<i>Meander</i>)	An object of the meander representation, implemented in <i>Meander.java</i> .
<i>tour</i> (<i>GridTours</i>)	An object of the grid tour representation, implemented in <i>GridTours.java</i> .
<i>mainStage</i> (<i>Stage</i>)	The main stage that the entire program is displayed/run in.
<i>mainScene</i> (<i>Scene</i>)	The scene includes the components that are necessary for visual aesthetics. The scene is added into the stage and the stage is displayed.

GENERATING THE PERMUTATION: *GeneratePermutation.java* (Class)

(DATA STRUCTURES PORTION)

<i>enterLabel</i> (<i>Label</i>)	<p>These are fixed labels that tell the user what the program is expecting:</p>  <p>User can either enter his own permutation or specify a new length to generate a random permutation (done by program in the background.)</p>
<i>randomLabel</i> (<i>Label</i>)	
<i>enterText</i> (<i>TextField</i>)	<p>These two text fields provide user the space to enter his desired input.</p>  <p>Note that each field is equipped with some sort of 'hint' for the user in order to make him abide by the rules.</p>
<i>randomText</i> (<i>TextField</i>)	

<i>enterYourOwn</i> (Button)	Click on 'Enter' after filling in the desired input.
<i>generateRandomly</i> (Button)	
<i>hbox1</i> (HBox) <i>hbox2</i> (HBox)	Each of the three things in the two sets of fields in the above screenshot are the represented using HBoxes .
<i>dialogBox</i> (VBox)	The two fields are represented using a VBox .
<i>integers []</i> (int)	In case the user entered the permutation himself, then the resulting numbers are stored within this array. This array is passed into a method to determine if the user abided by the rules: <ul style="list-style-type: none"> a. The length of permutation should be from 2 to 12 (inclusive). b. A permutation of length <i>N</i> should include distinct integers from 1 to <i>N</i>.
<i>size</i> (int)	The size that user enters for a random permutation is stored in this variable. This number is passed through a method to determine if the user a valid number and not random symbols/characters/letters. Also, the number should be from 2 to 12 (inclusive).
<i>stage</i> (Stage)	The window where all of the content exists.
<i>scene</i> (Scene)	The scene where all of the visual components are displayed.

DISPLAY SQUARE GRID REPRESENTATION: <i>SquareGrid.java</i> (Class) (DATA STRUCTURES PORTION)	
<i>sizeOfCellWidth</i> (int) <i>sizeOfCellHeight</i> (int)	When first experimenting with the size of each cell (square), it's troublesome to change each value of the square's length of sides. So, a variable was used. Both of the variables have the same number because it's a square, but two separate variables were used (in the beginning) in case each partition doesn't support a 12 x 12 grid.

<i>boardWidth</i> (<i>int</i>) <i>boardHeight</i> (<i>int</i>)	The grid height and width is set to dimensions that will allow even the largest grid possible to fit comfortably.
<i>cellsX</i> (<i>int</i>)	The number of cells in the <i>x</i> direction is determined by: $cellsX = boardWidth/sizeOfCellWidth;$
<i>cellsY</i> (<i>int</i>)	Similarly, the number of cells in the <i>y</i> direction is determined by: $cellsY = boardHeight/sizeOfCellHeight;$
<i>grid</i> <code>[] []</code> (<i>NxNGrid</i>)	Represents a 2D grid format of the entire square grid that will store information about the status of each cell. Here, <i>NxNGrid</i> refers to a constructor inside of a static class inside of this file. It's explained better in the next section, but this constructor simply designs the nature of the cell by assigning a color, border, stroke, etc.
<i>pane</i> (<i>Pane</i>)	The <i>Pane</i> is used to display the grid on the correct canvas. (Wherever the user wants)
<i>helpContent</i> (<i>Text</i>)	This will provide help for the user regarding this particular representation.
<i>hbox</i> (<i>HBox</i>)	This box adds the square grid representation and the help content as its children.
<i>pane2</i> (<i>Pane</i>)	This is the pane that is returned to the main program. <i>pane2</i> consists of <i>pane</i> and <i>hbox</i> as its children in that order.
<i>border</i> (<i>Rectangle</i>)	In this case, the rectangle's dimensions are <i>sizeOfCellWidth</i> and <i>sizeOfCellHeight</i> . Each cell is represented by a square. This square is defined by <i>border</i> .
<i>colorRow</i> (<i>int</i>)	Colored squares are represented in red and inversions are represented in green . The row that each colored square is represented through this variable. For a permutation of length <i>N</i> , a loop will run and modify this variable <i>N</i> times.

DISPLAY INVERSE REPRESENTATION: <i>Inverse.java</i> (Class) (DATA STRUCTURES PORTION)	
<i>inverse</i> (ArrayList)	Stores the inverse in an <i>ArrayList</i> .
<i>dummy</i> (String)	This is used to represent the inverse in a <i>String</i> format.
<i>inverseString</i> (Label)	The string from above is modified to a <i>Label</i> .
<i>helpContent</i> (Text)	This will provide help for the user regarding this particular representation.
<i>vbox</i> (VBox)	This box adds the inverse (<i>inverseString</i>) representation and the help content as its children.
<i>pane1</i> (Pane) <i>pane2</i> (Pane) <i>pane3</i> (Pane) <i>pane4</i> (Pane)	In whichever partition the user wants to see the inverse in, that partition will have a new pane defined and the inverse will be added to that pane.

DISPLAY INVERSION TABLE REPRESENTATION: <i>InversionTable.java</i> (Class) (DATA STRUCTURES PORTION)	
<i>inversion</i> [] (int)	The inversion of the permutation will be stored here.
<i>count</i> (int)	<p>This is a local variable and it's run <i>N</i> times through a loop. The logic of this variable is defined as follows:</p> <p><i>If the previous numbers from 1 to i are greater than the number in the correct index, then increment count by 1.</i></p>
<i>helpContent</i> (Text)	<p>These variables are already explained in <i>Inverse.java</i>. The definitions and implementations are exactly the same, except that <i>dummy</i> converts the inversion into a <i>String</i> and <i>inversionString</i> converts this <i>String</i> into a <i>Label</i>.</p>
<i>vbox</i> (VBox)	
<i>dummy</i> (String)	

<i>inversionString</i> (Label)	
<i>pane1</i> (Pane)	
<i>pane2</i> (Pane)	
<i>pane3</i> (Pane)	
<i>pane4</i> (Pane)	

DISPLAY FACTORADIC NUMBER REPRESENTATION: <i>FactoradicNumber.java</i> (Class)	
(DATA STRUCTURES PORTION)	
<i>inversion</i> (InversionTable)	An object of <i>InversionTable.java</i> and is used to call the inversion of permutation.
<i>permutationInversion</i> [] (int)	This variable is initialized by calling the inversion method in the above file. Basically, it stores the permutation's inversion.
<i>factorialStartingSize</i> (int)	The factoradic number will be calculated using factorials, which will be calculated through the (<i>permutation's size</i> – 1).
<i>factoradic</i> (int)	Initialized to zero in the beginning, this stores the permutation's factoradic representation.
<i>dummy</i> (String)	These variables are already explained in <i>Inverse.java</i> . The definitions and implementations are exactly the same, except that <i>dummy</i> converts the factoradic number into a <i>String</i> and <i>factoradicString</i> converts this <i>String</i> into a <i>Label</i> .
<i>factoradicString</i> (Label)	
<i>pane1</i> (Pane)	
<i>pane2</i> (Pane)	
<i>pane3</i> (Pane)	
<i>pane4</i> (Pane)	

DISPLAY PERMUTATION GRAPH REPRESENTATION: *GraphThePermutation.java*
(Class)

(DATA STRUCTURES PORTION)

<i>pane</i> (<i>Pane</i>)	A layout for displaying the meander representation onto the correct partition.
<i>nodes</i> <i>[]</i> <i>[]</i> (<i>PermutationGraph</i>)	The nodes for the graph representation are stored in this array.
<i>circle</i> (<i>Circle</i>)	Each node is drawn through a circle of a fixed radius.
<i>text</i> (<i>Text</i>)	Each node will have the digit corresponding to it.
<i>stack</i> (<i>StackPane</i>)	The node's number and the node are displayed using a StackPane where the number is on top of the node.
<i>squares</i> (<i>SquareGrid</i>)	The graph representation takes help of the square grid representation to solve the problem.
<i>count</i> (<i>int</i>)	Not necessarily for data structures, but more for verification of the algorithm; This variable just makes sure that the total number of edges drawn is equal to the number of <i>green</i> squares in the square grid representation.
<i>edge</i> (<i>Path</i>)	For drawing the path/edge from one node to another
<i>x1</i> (<i>int</i>) <i>y1</i> (<i>int</i>) <i>x2</i> (<i>int</i>) <i>y2</i> (<i>int</i>)	(<i>x1</i> , <i>y1</i>) determines the center of one of the nodes. (<i>x2</i> , <i>y2</i>) determines the center of another one of the nodes.
<i>move</i> (<i>MoveTo</i>)	For setting the coordinates for one of the nodes.
<i>quad</i> (<i>QuadCurveTo</i>)	For setting the coordinates for another one of the nodes. The reason that a quadratic curve was used is because since the screen is partitioned into four areas, the permutation graph will look like a mess if there the permutation length is too large.

	In that case, using lines as edges did not appear to be feasible. Therefore, a Bezier quadratic curve was used to draw each edge.
--	---

DISPLAY MEANDER REPRESENTATION: <i>Meanders.java</i> (Class) (DATA STRUCTURES PORTION)	
<i>pane</i> (<i>Pane</i>)	A layout for displaying the meander representation onto the correct partition.
<i>nodes</i> [] [] (<i>Meander</i>)	The nodes for the meander representation are stored in this array.
<i>circle</i> (<i>Circle</i>)	Each node is drawn through a circle of a fixed radius.
<i>text</i> (<i>Text</i>)	Each node will have the digit corresponding to it.
<i>stack</i> (<i>StackPane</i>)	The node's number and the node are displayed using a StackPane where the number is on top of the node.
<i>setOfPaths</i> [] (<i>Path</i>)	The paths from each node to another is displayed using an array of type <i>Path</i> .

DISPLAY GRID TOUR REPRESENTATION: <i>GridTours.java</i> (Class) (DATA STRUCTURES PORTION)	
<i>sizeOfCellWidth</i> (<i>int</i>) <i>sizeOfCellHeight</i> (<i>int</i>)	When first experimenting with the size of each cell (square), it's troublesome to change each value of the square's length of sides. So, a variable was used. Both of the variables have the same number because it's a square, but two separate variables were used (in the beginning) in case each partition doesn't support a 12 x 12 grid.
<i>boardWidth</i> (<i>int</i>) <i>boardHeight</i> (<i>int</i>)	The grid height and width are set to dimensions that will allow even the largest grid possible to fit comfortably.
<i>cellsX</i> (<i>int</i>)	The number of cells in the <i>x</i> direction is determined by: <i>cellsX</i> = <i>boardWidth</i> / <i>sizeOfCellWidth</i> ;

<i>cellsY</i> (<i>int</i>)	Similarly, the number of cells in the y direction is determined by: <i>cellsY = boardHeight/sizeOfCellHeight;</i>
<i>grid</i> [] [] (<i>NxNGrid</i>)	Represents a 2D grid format of the entire square grid that will store information about the status of each cell. Here, <i>NxNGrid</i> refers to a constructor inside of a static class inside of this file. It's explained better in the next section, but this constructor simply designs the nature of the cell by assigning a color, border, stroke, etc.
<i>pane</i> (<i>Pane</i>)	The <i>Pane</i> is used to display the grid on the correct canvas. (Wherever the user wants)
<i>helpContent</i> (<i>Text</i>)	This will provide help for the user regarding this particular representation.
<i>hbox</i> (<i>HBox</i>)	This box adds the square grid representation and the help content as its children.
<i>pane2</i> (<i>Pane</i>)	This is the pane that is returned to the main program. <i>pane2</i> consists of <i>pane</i> and <i>hbox</i> as its children in that order.
<i>array</i> [] (<i>int</i>)	To store the second permutation's values my index
<i>rows</i> (<i>ArrayList</i>) <i>columns</i> (<i>ArrayList</i>)	To determine which rows and columns still need to be traversed since only one traversal is possible.
<i>visitedRows</i> (<i>ArrayList</i>) <i>visitedColumns</i> (<i>ArrayList</i>)	Keep a list of visited rows and columns, just in case they're needed.
<i>currentRow</i> (<i>int</i>) <i>currentColumn</i> (<i>int</i>)	Keep a track of the current position of the program in the grid, with respect to row and column.
<i>visitedRow</i> (<i>int</i>) <i>visitedColumn</i> (<i>int</i>)	Keep a track of the previously visited position of the program in the grid, with respect to row and column.

IMPORTANT METHODS AND CLASSES

Since there are many files, each file's methods will be represented separately from others. (Each file contains a single class.) This should clear any unneeded confusion and hopefully, provide better clarity.

Note: The default methods that Java requires to be implemented (but are not implemented in this project) are not shown.

MAIN PROGRAM: <i>MainProgram.java (Class)</i>	
<i>main</i> () (<i>Method of no return type</i>)	The method that launches the JavaFX application.
<i>start</i> () (<i>Method of no return type</i>)	Implement the visual components and their action listeners in this method.
<i>initializeSplitPanels</i> (<i>Method of VBox return type</i>)	Divide the empty canvas into five partitions (four for displaying representations and one for entering/generating a permutation).
<i>menuOptions</i> (<i>Method of no return type</i>)	Display the menu options available for representing a permutation. Add action listeners to each of the representations. (<i>All seven are implemented in seven separate, corresponding files.</i>)
<i>displayErrorMessage</i> (<i>Method of no return type</i>)	Create a new stage and display an error message on it every time a user does something that violates a condition/constraint.
<i>resetPreviousRepresentations()</i> (<i>Method of no return type</i>)	Whenever a new permutation is generated/entered, then all the partitions are cleared of the representations from the previous permutation.

GENERATING THE PERMUTATION: <i>GeneratePermutation.java (Class)</i>	
<i>popUpWindow</i> () (<i>Method of no return type</i>)	Generate a pop-up window that asks the user to either enter a permutation or enter a length for a random permutation.

<i>checkOwnPermutationFields</i> (Method of Boolean return type)	Check if the entered permutation satisfies the conditions: <ul style="list-style-type: none"> a. Permutation length must be from 2 to 12 (inclusive) only. b. A permutation of length N should contain distinct integers from 1 to N. c. Each digit should be separated by a space; otherwise, it's considered to be a single number.
<i>checkRandomPermutationFields</i> (Method of Boolean return type)	Check if the entered length of random permutation is between 2 and 12 (inclusive). Also, generate an error message if user enters anything other than a number such as symbols/characters/etc.
<i>cyclicNotation</i> (Method of ArrayList return type)	For representing the permutation in its corresponding cyclic notation. (This method is called if and only a valid permutation is generated.)
<i>displayErrorMessages</i> (Method of no return type)	Create a new stage and display an error message on it every time a user does something that violates a condition/constraint.

DISPLAY SQUARE GRID REPRESENTATION: <i>SquareGrid.java</i> (Class)	
<i>removePreviousRepresentations</i> () (Method of no return type)	If the partition has another representation in it, then it will be removed.
<i>NxNGrid</i> (Static class)	This draws an $N \times N$ grid where N is the length of the permutation.
<i>NxNGrid</i> () (Constructor for the above class)	This constructor defines each square's default color, border thickness, etc.
<i>createTable</i> (Method of Pane return type)	This is responsible for adding the grid to the pane. It will call the below two methods.
<i>coloredSquares</i> () (Method of no return type)	The colored squares are denoted with red .

<i>inversions</i> () (Method of no return type)	The inversions are denoted with green .
--	--

DISPLAY INVERSE REPRESENTATION: <i>Inverse.java</i> (Class)	
<i>findingInverse</i> () (Method of no return type)	<p>The permutation's inverse representation is calculated inside this method.</p> <p>Since the code inside this method is relatively short, a separate method for removing previous representation is not created.</p>

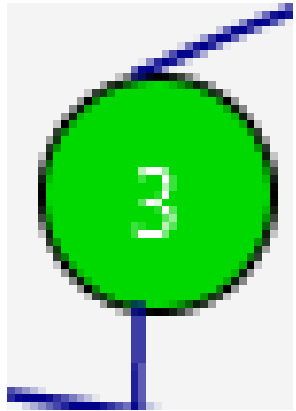
DISPLAY INVERSION TABLE REPRESENTATION: <i>InversionTable.java</i> (Class)	
<i>inversion</i> () (Method of no return type)	<p>The permutation's inversion table representation is calculated inside this method.</p> <p>Since the code inside this method is relatively short, a separate method for removing previous representation is not created.</p>
<i>inversion</i> () (Method of int [] return type)	<p>(Method overloading)</p> <p>This method is used when the factoradic number needs to be calculated. In that case, the inversion is returned as an array of type int.</p>

DISPLAY FACTORADIC REPRESENTATION: <i>FactoradicNumber.java</i> (Class)	
<i>factoradic</i> () (Method of no return type)	<p>The permutation's factoradic number representation is calculated inside this method.</p> <p>Since the code inside this method is relatively short, a separate method for removing previous representation is not created.</p>
<i>factorial</i> () (Method of int return type)	Calculate the factorial inside this method.

DISPLAY PERMUTATION GRAPH REPRESENTATION: <i>GraphThePermutation.java</i> (Class)
--

<i>removePreviousRepresentations</i> <i>() (Method of no return type)</i>	If the partition has another representation in it, then it will be removed.
<i>PermutationGraph</i> <i>(Static class)</i>	For designing the structure of how the graph will appear in each partition.
<i>PermutationGraph (int, int)</i> <i>PermutationGraph (int, int, int)</i> <i>(Overloaded constructors for the above class)</i>	<p>These two are constructors for the above class.</p> <p>The first constructor draws the node and specifies it with a color.</p> <p>The second constructor is used to label that node with the vertex number.</p> <div data-bbox="904 743 1153 957" data-label="Image"> </div> <p>In the above picture, the bluish-black circle is drawn by the first constructor and the number '12' is displayed by the second constructor.</p> <p>Obviously, the number of nodes will depend on the permutation's size.</p>
<i>createTable</i> <i>() (Method of Pane return type)</i>	This method will display the graph above in the appropriate panes.
<i>inversions</i> <i>() (Method of Path return type)</i>	This method will be responsible for drawing the actual edges between the nodes and return this set to the method above.

DISPLAY MEANDER REPRESENTATION: <i>Meanders.java (Class)</i>	
<i>removePreviousRepresentations</i> <i>() (Method of no return type)</i>	If the partition has another representation in it, then it will be removed.
<i>Meander</i> <i>(Static class)</i>	For designing the structure of how the graph will appear in each partition.

<p><i>Meander (int, int)</i></p> <p><i>Meander (int, int, int)</i></p> <p><i>(Overloaded constructors for the above class)</i></p>	<p>These two are constructors for the above class.</p> <p>The first constructor draws the node and specifies it with a color.</p> <p>The second constructor is used to label that node with the vertex number.</p>  <p><i>(Please ignore the lines. This was a screenshot of a representation of a randomly generated permutation.)</i></p> <p>In the above picture, the bluish-black circle is drawn by the first constructor and the number '12' is displayed by the second constructor.</p> <p>Obviously, the number of nodes will depend on the permutation's size.</p>
<p><i>createMeander ()</i> <i>(Method of Pane return type)</i></p>	<p>Display the meander onto the screen and correct partition.</p>
<p><i>quadraticCurves ()</i> <i>(Method of Path[] return type)</i></p>	<p>Draw edges within the meander by calling the data from the main program's permutation.</p>

DISPLAY GRID TOUR REPRESENTATION: <i>GridTours.java</i> (Class)	
<p><i>removePreviousRepresentations ()</i> <i>(Method of no return type)</i></p>	<p>If the partition has another representation in it, then it will be removed.</p>

<i>NxNGrid</i> (Static class)	This draws an $N \times N$ grid where N is the length of the permutation.
<i>NxNGrid</i> () (Constructor for the above class)	This constructor defines each square's default color, border thickness, etc.
<i>createGrid</i> () (Method of <i>Pane</i> return type)	Display the grid onto the screen and in the correct partition.
<i>incomplete</i> () (Method of <i>Boolean</i> return type)	Determine if there are still rows/columns to be traversed (return <i>true</i> , if so).
<i>complete</i> () (Method of <i>Boolean</i> return type)	Determine if there are still rows/columns to be traversed (return <i>false</i> , if so).
<i>colorGrid</i> () (Method of <i>int[]</i> return type)	Draw the grid tour onto the grid.
<i>checkForRightTurns</i> () (Method of <i>Boolean</i> return type)	ONLY allow the program to take right turns. These right turns are defined within this method.
<i>redSquares</i> () (Method of no return type)	Wherever there is a right turn, mark a red square, just for the user to be able to see the path.
<i>displaySecondPermutation</i> () (Method of <i>String</i> return type)	Convert the array of integers of the second permutation into a string before returning it to the calling method.

INTERNAL DATA STRUCTURES

STORAGE

There are two types of storage in this project. One is for storing the actual visuals of the project. And the other was storing the data structures necessary to run the algorithm without the user's obvious knowledge of them.

For the visuals:

At first, it seemed effective to use linked lists to draw grid tours. It somehow made sense. But it was later rejected with much thought because the time complexity was too big. The accumulating heap memory would be impossible to ignore since the user's interaction will be affected through bad performance. *Plus*, it was against the project's rules to use *ANY* type of link.

A simpler data structure was the usage of a 2-D array of Split Panes. It just seemed easier and less costly to store the partitions within an array, where each element of the array corresponds to a single partition. There are five partitions in total and the center one is already fixed. There's no changing that (except during permutation generation). All throughout the program, for the most part, the partitions' contents will be changed. Therefore, a 2D array seemed effective.

For the data structures:

For the square grid, a 2D array was used. For inversion, inverse, and factoradic, since they're all related to a type of arithmetic, ArrayLists were used for all of them.

For permutation graphs, meanders, and grid tours, graphs were used. These graphs were individually represented using dynamically growing ArrayLists for each one of them.

DATA STRUCTURES

About 80-90% of the visuals are stored in MainProgram.java and GeneratePermutation.java.

The rest of the files included purely algorithms and data structures. (There may be a few lines of visuals here and there, but only necessary ones are incorporated within the same file as the data structures.)

GRAPH

Especially, for permutation graph, meanders, and grid tours, the concept of a graph was used. For a permutation graph, an edge's endpoints (vertices) have to be determined using the graph concept. For meanders, a graph is used to check if the edges intersect or not. For a grid tour, it's used to check if the program is making the correct turns or not.

2D-ARRAYS

This type of data structure is used especially for the square grid representation. Inverse, inversion, and factoradic representations were calculated normally and didn't require any complex data structures. A square grid, however, did and also posed an N^2 complexity because the grid is of size $N \times N$.

REPRESENTATIONS

The visuals for the representations are provided in greater detail in the next section. This section only discusses about the coding aspects of the representations.

Also, each representation is defined in a new class and treated as a separate object, independent of anything else. This new class is defined in a new *file* as well. In some of the previous projects, many classes were defined as *static* to make them reusable in a single file. But using multiple files seemed to be beneficial in terms of better structural organization.

Also, before any representation is drawn, the program checks for any existing representations present in the partition already. If so, then it's removed *first*. The removal of the representations is defined in a separate method for some of them. If the code is long for any one of the representations, then a separate method was used. Overall, the code doesn't change:

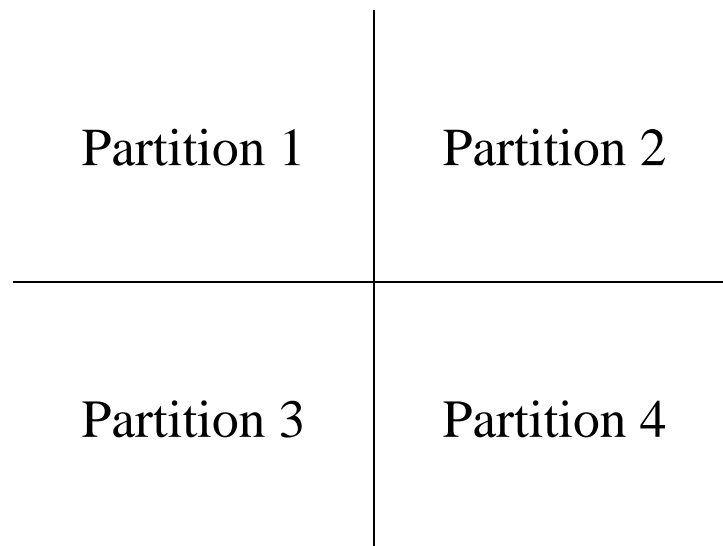
```

public void findInverse(int whichMenu) {
    try {
        // delete the previous representation (whatever it is)
        switch (whichMenu) {
            case 1:
                MainProgram.canvas1.getItems().remove(1);
                break;
            case 2:
                MainProgram.canvas2.getItems().remove(1);
                break;
            case 3:
                MainProgram.canvas3.getItems().remove(1);
                break;
            case 4:
                MainProgram.canvas4.getItems().remove(1);
                break;
        }
    } catch (Exception e) {
        // ignore

        /*
         * This exception is raised when there is initially nothing present (index out
         * of bounds exception)
         */
    }
}

```

No matter how the method is defined, the code above remains unchanged. The variable *whichMenu* (*int*) represents which partition the user clicked on. To understand how the partitions are drawn, the following figure is drawn for convenience:



SQUARE GRID REPRESENTATION

The square grid was one of the easiest representations to do with respect to algorithm, but a bit challenging when it came to graphical presentations.

Internally, the grid was modified/changed/defined through a 2D array. According to the definition of a grid representation, the grid's internal data was changed using iterations. The number of iterations will match the length of the permutation – $O(n)$.

Next, the display of the grid was managed through panes. Each partition is seen as a separate window and independent of anything outside the boundary. Therefore, the program first has to determine where the user clicked and which menu he accessed. This is done by accessing mouse event listeners.

A method is then called to read the permutation (by this time, a valid permutation would've been on the screen already) and determine which squares to color and which squares to leave alone. Then this data is passed to a method that manages the panes. In whichever pane the user wants the data, that pane is redrawn (previous representation is removed) with this representation.

The code for that is displayed below:

```
import javafx.geometry.Insets;

public class SquareGrid {

    static final int sizeOfCellWidth = 22, sizeOfCellHeight = 22, boardWidth = 275, boardHeight = 275;
    static int cellsX = boardWidth / sizeOfCellWidth;
    static int cellsY = boardHeight / sizeOfCellHeight;
    public static NxNGrid[][] grid = new NxNGrid[cellsX][cellsY];
    Pane pane;

    public void removePreviousRepresentation(int whichMenu) {
        cellsX = MainProgram.permutation.size();
        cellsY = MainProgram.permutation.size();
        try {
            // delete the previous representation (whatever it is)
            switch (whichMenu) {
                case 1:
                    MainProgram.canvas1.getItems().remove(1);
                    break;
                case 2:
                    MainProgram.canvas2.getItems().remove(1);
                    break;
                case 3:
                    MainProgram.canvas3.getItems().remove(1);
                    break;
                case 4:
                    MainProgram.canvas4.getItems().remove(1);
                    break;
            }
        } catch (Exception e) {
            // ignore
        }
        /*
         * This exception is raised when there is initially nothing present (index out
         * of bounds exception)
         */
    }
}
```



```

static class NxNGrid extends StackPane {

    // each rectangle is defined as a square.
    // A square = 1 cell;
    // A grid = (cellsX) * (cellsY) number of squares/cells

    Rectangle border = new Rectangle(sizeOfCellWidth, sizeOfCellHeight);

    // This constructor is called each time a new cell has to be defined
    public NxNGrid(int x, int y) {
        // Add each cell as the pane's children
        getChildren().add(border);
        // Each cell's dimensions are of sizeOfCellWidth and sizeOfCellHeight
        setTranslateX(x * sizeOfCellWidth);
        setTranslateY(y * sizeOfCellHeight);
        // Each cell is initially of a light blue color
        border.setFill(Color.LIGHTGREY);
        border.setStroke(Color.BLACK);
    }
}

public Pane createTable() {
    pane = new Pane();
    pane.setPadding(new Insets(10));
    // Define board's dimensions
    pane.setPrefSize(boardWidth, boardHeight);
    // Fill the board with cells
    for (int i = 0; i < cellsX; i++) {
        for (int j = 0; j < cellsY; j++) {
            grid[i][j] = new NxNGrid(i, j);
            pane.getChildren().add(grid[i][j]);
        }
    }
}

coloredSquares(); // red ones

```

```

coloredSquares(); // red ones
inversions(); // green ones
Pane pane2 = new Pane();
HBox hbox = new HBox();
Text helpText = new Text("The square grid on the left shows one of the representations for a \n"
    + "permutation. \n\nThe rules are that for a permutation of size N, a grid \n"
    + "of NxN dimensions must be created first.\n\n"
    + "Column j of row i is colored RED whenever permutation[i]=j.\n\n"
    + "Then, GREEN colored squares represent inversions:\n"
    + "These are colored by seeing those squares which have colored \n"
    + "squares lying both below (in same column) and to their right (same \n"
    + "row). If there are no GREEN squares, then the permutation is ordered.");
hbox.getChildren().addAll(pane, helpText);
pane2.getChildren().addAll(hbox);
// Return pane filled with children
return pane2;
}

public void coloredSquares() {
    // coloring those squares that will not be part of set of inversions
    for (int i = 0; i < MainProgram.permutation.size(); i++) {
        // horizontally coloring squares
        int colorRow = MainProgram.permutation.get(i);
        for (int j = colorRow; j < cellsY; j++) {
            grid[j][i].border.setFill(Color.WHITE);
        }
        // vertically coloring squares
        for (int j = i; j < cellsY; j++) {
            grid[colorRow - 1][j].border.setFill(Color.WHITE);
        }
    }

    for (int i = 0; i < MainProgram.permutation.size(); i++) {
        int colorRow = MainProgram.permutation.get(i);
        /*
        * Remember that the permutation's 'row' is from 1 to N. However, an ArrayList's
        * indices are from 0 to N-1
        */
        grid[colorRow - 1][i].border.setFill(Color.RED);
    }
}

// remaining squares are inversions
public void inversions() {
    for (int i = 0; i < cellsX; i++) {
        for (int j = 0; j < cellsY; j++) {
            if (grid[i][j].border.getFill().equals(Color.LIGHTGREY))
                grid[i][j].border.setFill(Color.LIME);
        }
    }
}
}

```

The explanation along with the figure is explained to the user *beside* the figure itself. To avoid redundancy, the detailed picture of the output of the representation is shown in the screen's visuals.

INVERSE

The first idea to find the inverse was to just use the square grid. But that proved to have about a time complexity of $O(n^2)$ because any object that is instantiated from the square grid will have a time complexity of *at least* $O(n^2)$.

Drawing the grid itself was of quadratic complexity since $N \times N$ squares are required.

Therefore, with help of class notes, the inverse was found using the normal, mathematical way by lining up two permutations – one sorted already and the other of the original permutation.

I used an *ArrayList*, which is like an enhanced version of an array. *ArrayLists* are not linked lists and don't have any links, but they do provide better access methods than a conventional array. The code for the inverse is as follows:

(The code for removing previous representations is not shown here to avoid additional redundancy.)

```
ArrayList<Integer> inverse = new ArrayList<Integer>();

for (int i = 1; i <= MainProgram.permutation.size(); i++)
    inverse.add(MainProgram.permutation.indexOf(i) + 1);

String dummy = new String("INVERSE: ");
for (int i = 0; i < inverse.size(); i++) {
    dummy += " " + inverse.get(i);
}

System.out.println(dummy);
Label inverseString = new Label(dummy);
inverseString.setTextFill(Color.DARKBLUE);
inverseString.setFont(Font.font("Times New Roman", FontWeight.BOLD, 35));
Text helpText = new Text(
    "\n\n\tA permutation's inverse is another permutation of the same length with the same number of elements.\n"
    + "\n\tTo find the inverse, first list out two rows, with one top lining up exactly on top of another.\n"
    + "\n\tThe top row should be sorted, whereas the bottom resembles the original permutation.\n"
    + "\n\tInterchange the two rows and sort top new row in increasing order using the second one.\n"
    + "\n\tFor example, say the first digit of top row is 10. And the first digit of bottom row is 5.\n"
    + "\n\tIn this case, the digit 10 would be in the 5th position of the inverse.");

VBox vbox = new VBox();
vbox.getChildren().addAll(inverseString, helpText);
// add this new representation into the correct partition
switch (whichMenu) {
    case 1:
        Pane pane1 = new Pane();
        pane1.getChildren().add(vbox);
        MainProgram.canvas1.getItems().add(pane1);
        break;
    case 2:
```

```
        break;
    case 2:
        Pane pane2 = new Pane();
        pane2.getChildren().add(vbox);
        MainProgram.canvas2.getItems().add(pane2);
        break;
    case 3:
        Pane pane3 = new Pane();
        pane3.getChildren().add(vbox);
        MainProgram.canvas3.getItems().add(pane3);
        break;
    case 4:
        Pane pane4 = new Pane();
        pane4.getChildren().add(vbox);
        MainProgram.canvas4.getItems().add(pane4);
        break;
}
}
```

The *switch-case {}* is for displaying the inverse, along with the help content onto the correct pane.

INVERSION TABLE

Inversion table is also found similarly, but this time an *array* is used. The reason that an *ArrayList* was not used is because an *ArrayList* supports dynamic additions and deletions. An array does *not*. Since an inversion table is of the same length as the permutation itself, a static memory is needed. And the data structure required for that is an *array*.

A simple loop of size N was required to check the condition for an inversion table and this took $O(n)$ time complexity.

(The code for removing previous representations AND adding that representation onto the correct pane is not shown here to avoid additional redundancy. ONLY the internal, external representations are shown here.)

```
int[] inversion = new int[MainProgram.permutation.size()];

for (int i = 0; i < MainProgram.permutation.size(); i++) {
    int count = 0;
    for (int j = 0; j < i; j++) {
        /*
         * if the previous numbers from 1 to i are greater than the number in the
         * correct index, then do count++;
         */
        if (MainProgram.permutation.get(j) > MainProgram.permutation.get(i))
            count++;
    }
    // add this value of count variable into the inversion table
    System.out.println(count);
    inversion[MainProgram.permutation.get(i) - 1] = count;
}

String dummy = new String("INVERSION TABLE: ");
for (int i = 0; i < inversion.length; i++) {
    dummy += " " + inversion[i];
}

System.out.println(dummy);
Label inversionString = new Label(dummy);
inversionString.setTextFill(Color.DARKBLUE);
inversionString.setFont(Font.font("Times New Roman", FontWeight.BOLD, 35));
Text helpText = new Text("\n\n\tAn inversion table uniquely determines the corresponding permutation.\n"
    + "\n\tInversion table b1, b2, b3, ..., bN of the permutation a1, a2, a3, ..., aN is obtained by letting bj \n"
    + "\tbe the number of elements to the left of ai = j greater than j. \n"
    + "\t\t(i.e., bj = number of inversions whose second component equals j.)");

VBox vbox = new VBox();
vbox.getChildren().addAll(inversionString, helpText);
// add this new representation into the correct partition
switch (whichMenu) {
    case 1:
        Page page1 = new Page();
```

FACTORADIC NUMBER

Factoradic number was calculated using inversion table. An array was needed since the return type for an inversion table is an array. Not only that, the reason that the array was not converted into an *ArrayList* is because an *ArrayList* supports dynamic additions and deletions. An array does *not*. Since an inversion table is of the same length as the permutation itself, a static memory is needed. And the data structure required for that is an *array*.

```

import javafx.scene.control.Label;

public class FactoradicNumber {
    public void factoradic(int whichMenu) {
        try {
            // delete the previous representation (whatever it is)
            switch (whichMenu) {
                case 1:
                    MainProgram.canvas1.getItems().remove(1);
                    break;
                case 2:
                    MainProgram.canvas2.getItems().remove(1);
                    break;
                case 3:
                    MainProgram.canvas3.getItems().remove(1);
                    break;
                case 4:
                    MainProgram.canvas4.getItems().remove(1);
                    break;
            }
        } catch (Exception e) {
            // ignore

            /*
             * This exception is raised when there is initially nothing present (index out
             * of bounds exception)
             */
        }

        InversionTable inversion = new InversionTable();
        int[] permutationInversion = inversion.inversion();
        int factorialStartingSize = MainProgram.permutation.size() - 1;
        String dummy = new String("FACTORADIC REPRESENTATION: \n");

        long factoradic = 0;
        for (int i = factorialStartingSize, j = 0; i >= 0; i--, j++) {
            int result = factorial(i);

```

```

long factoradic = 0;
for (int i = factorialStartingSize, j = 0; i >= 0; i--, j++) {
    int result = factorial(i);
    factoradic += permutationInversion[j] * result;
}

dummy += " " + factoradic;

System.out.println(dummy);
Label factoradicString = new Label(dummy);
factoradicString.setTextFill(Color.DARKBLUE);
factoradicString.setFont(Font.font("Times New Roman", FontWeight.BOLD, 35));

String inversionTable = new String();
for (int i = 0; i < permutationInversion.length; i++) {
    inversionTable += Integer.toString(permutationInversion[i]) + " ";
}
String explanation = construction(factorialStartingSize, permutationInversion);
Text helpText = new Text(
    "\n\tA factoradic number is a sequence of digits where each digit has a different radix in particular, the radix \n"
    + "\tfor digit dI = (n-I)!, where I = index and N = length of permutation."
    + "\n\ti.e., rightmost digit has base!, next digit has base 2!, and so on such that each digit\n"
    + "\tin a factoradic number is smaller than its base.\n"
    + "\n\tFirst, find the inversion table of the given permutation: " + inversionTable
    + "\n\tNext, calculate the factoradic number: " + "\n\t\t" + explanation + " = " + factoradic);
VBox vbox = new VBox();
vbox.getChildren().addAll(factoradicString, helpText);

// add this new representation into the correct partition
switch (whichMenu) {
case 1:
    Pane panel1 = new Pane();
    panel1.getChildren().add(vbox);
    MainProgram.canvas1.getItems().add(panel1);
    break;
case 2:

```

```

    pane2.getChildren().add(vbox);
    MainProgram.canvas2.getItems().add(pane2);
    break;
case 3:
    Pane pane3 = new Pane();
    pane3.getChildren().add(vbox);
    MainProgram.canvas3.getItems().add(pane3);
    break;
case 4:
    Pane pane4 = new Pane();
    pane4.getChildren().add(vbox);
    MainProgram.canvas4.getItems().add(pane4);
    break;
}
}

private int factorial(int i) {
    int result = 1;
    for (int j = 2; j <= i; j++)
        result *= j;
    return result;
}

//for showing the calculation of factoradic number
private String construction(int size, int[] inversion) {
    String work = new String();
    for (int i = size - 1, j = 0; i >= 0 && j < inversion.length; i--, j++) {
        work += "( " + Integer.toString(inversion[j]) + " x " + Integer.toString(i) + " ! )";
        if (i == 4)
            work += "\n\t\t\t\t";
        if (i != 0)
            work += " + ";
    }
    return work;
}
}

```

The *LAST* method is responsible for showing the work of how the factoradic number is calculated. This screen visual is also displayed in the next section.

PERMUTATION GRAPH

A *graph* data structure was used as expected for the permutation graph. It was needed to determine how edges should be drawn.

In this case, edges are drawn visually by using coordinates, etc. But the data structure needed was the square grid representation to see where the inversions are. For each green colored square, an inversion is present. The row and column of the inversion marked the endpoints of each edge.

ermutationsProject ▸ src ▸ (default package) ▸ GraphThePermutation ▸

```
import java.util.Random;

public class GraphThePermutation {
    public void removePreviousRepresentation(int whichMenu) {
        try {
            // delete the previous representation (whatever it is)
            switch (whichMenu) {
                case 1:
                    MainProgram.canvas1.getItems().remove(1);
                    break;
                case 2:
                    MainProgram.canvas2.getItems().remove(1);
                    break;
                case 3:
                    MainProgram.canvas3.getItems().remove(1);
                    break;
                case 4:
                    MainProgram.canvas4.getItems().remove(1);
                    break;
            }
        } catch (Exception e) {
            // ignore

            /*
             * This exception is raised when there is initially nothing present (index out
             * of bounds exception)
             */
        }
    }

    Pane pane;
    public static PermutationGraph[][] nodes = new PermutationGraph[12][1];

    // private static Random randomColorPicker = new Random();

    static class PermutationGraph extends StackPane {
```



```

static class PermutationGraph extends StackPane {

    Circle circle = new Circle(15);
    // each circle = one node

    // This constructor is called each time a new node has to be defined
    public PermutationGraph(int x, int y) {
        // Add each cell as the pane's children
        getChildren().add(circle);
        int scale = x % 3;
        setTranslateX(scale * 150);
        setTranslateY((int) (x / 3) * 60);

        // Each cell is initially of a light blue color
        Color randomColorFill = Color.rgb(0, 0, 255 - x * 19);

        circle.setFill(randomColorFill);
        circle.setStroke(Color.BLACK);
    }

    Text text = new Text();

    // for numbering each node
    public PermutationGraph(int x, int y, int stroke) {
        getChildren().add(text);
        int scale = x % 3;
        setTranslateX(scale * 150);
        setTranslateY((int) (x / 3) * 60);
        text.setText(Integer.toString(x + 1));
        text.setFill(Color.WHITE);
        text.setStrokeWidth(stroke);
    }
}

```

```

public Pane createTable() {
    pane = new Pane();
    pane.setPadding(new Insets(10));
    // Define board's dimensions
    pane.setPrefSize(240, 240);
    // Fill the board with cells
    for (int i = 0; i < MainProgram.permutation.size(); i++) {
        for (int j = 0; j < 1; j++) {

            StackPane stack = new StackPane();

            nodes[i][j] = new PermutationGraph(i, j);

            PermutationGraph text = new PermutationGraph(i, j, 4);

            stack.getChildren().addAll(nodes[i][j], text);
            pane.getChildren().addAll(stack);

        }
    }

    SquareGrid squares = new SquareGrid();
    squares.createTable();
    int count = 0;
    for (int i = 0; i < MainProgram.permutation.size(); i++) {
        for (int j = 0; j < MainProgram.permutation.size(); j++) {
            if (squares.grid[i][j].border.getFill().equals(Color.LIME)) {
                Path inversionEdge = inversions(i, j);
                pane.getChildren().add(inversionEdge);
                count++;
            }
        }
    }
    System.out.println(count);
    return pane;
}

```

```

        return pane;
    }

// static int colorIncrement = 10, currentStatus = 100;

public Path inversions(int i, int j) { // i=1; j=1;
    // initializations
    System.out.println("i: " + i + "\tj:" + j);
    Path edge = new Path();

    int x1 = (MainProgram.permutation.get(j) % 3);
    int y1 = (int) Math.floor((MainProgram.permutation.get(j) + 1) / 3) - 1;

    int x2 = ((i) % 3);
    int y2 = (i) / 3;

    MoveTo move = new MoveTo();
    QuadCurveTo quad = new QuadCurveTo();
    // draw the edges
    move.setX(15 + (x1) * 150);
    move.setY(15 + 30 * 2 * y1 - 0);

    quad.setX(15 + x2 * 150);
    quad.setY(15 + 15 * y2 + 30 * y2);
    quad.setControlX(quad.getX());
    quad.setControlY(quad.getY());

    // edge.setFill(Color.BLACK);
    edge.getElements().addAll(move, quad);
    edge.setStroke(Color.BLACK);
    edge.setStrokeWidth(1.5);

    return edge;
}
}

```

MEANDER

The concept is very similar as that of the permutation graph.

BUT the visual representation is *very* different and dissimilar to the permutation graph. The data structure behind was similar to permutation graph because of these representations required knowing where inversions laid.

```

import javafx.geometry.Insets;

public class Meanders {

    public void removePreviousRepresentation(int whichMenu) {}

    Pane pane;
    public static Meander[][] nodes = new Meander[12][1];

    static class Meander extends StackPane {

        Circle circle = new Circle(15);
        // each circle = one node

        // This constructor is called each time a new circle has to be defined
        public Meander(int x, int y) {
            // Add each circle as the pane's children
            getChildren().add(circle);
            // Each cell's dimensions are of a fixed radius
            setTranslateX(x * 40);
            setTranslateY(y * 40);
            // Each circle is initially of a green color - changed for each node
            Color randomColorFill = Color.rgb(0, 255 - x * 19, 0);

            circle.setFill(randomColorFill);
            circle.setStroke(Color.BLACK);

        }

        Text text = new Text();

        public Meander(int x, int y, int stroke) {
            getChildren().add(text); // add text as a child
            setTranslateX(x * 40); // scaling each vertex number
            setTranslateY(y * 40);
            text.setText(Integer.toString(x + 1)); // parse the integer as a string
            text.setFill(Color.WHITE); // color the text
        }
    }
}

```

```

        text.setFill(Color.WHITE); // color the text
        text.setStrokeWidth(stroke); // make the text readable
    }
}

public Pane createMeander() {
    pane = new Pane();
    pane.setPadding(new Insets(10));
    // Define board's dimensions
    pane.setPrefSize(240, 240);
    // Fill the board with cells
    for (int i = 0; i < MainProgram.permutation.size(); i++) {
        for (int j = 0; j < 1; j++) {
            // add node and vertex name in one stack and return that stack
            StackPane stack = new StackPane();
            nodes[i][j] = new Meander(i, j);
            Meander text = new Meander(i, j, 4);
            stack.getChildren().addAll(nodes[i][j], text);
            // add stack as a child to pane
            pane.getChildren().add(stack);
        }
    }

    // meander edges
    Path setOfPaths[] = quadraticCurves(MainProgram.permutation.size());

    // add the returned edges into the pane.
    for (int i = 0; i < MainProgram.permutation.size(); i++) {
        try {
            pane.getChildren().add(setOfPaths[i]);
        } catch (NullPointerException e) {
            // ignore
        }
    }
}

```

```

// insert help content to be displayed beside the meander representation
Pane pane2 = new Pane();
HBox hbox = new HBox();
Text helpText = new Text("\n\n\n\t\tTake a straight line and number it in equal segments with \n"
    + "integers. There are two versions to every permutation: open and closed \n"
    + "meanders. We want closed. Closed meanders cycle back to starting place \n"
    + "without crisscrossing with other \"routes\" in the middle. \n"
    + "Left to right edges are colored BLUE. \n" + "Right to left edges are colored RED.");
hbox.getChildren().addAll(pane, helpText);
pane2.getChildren().add(hbox);
// Return pane filled with children
return pane2;
}

```

The following is the method for drawing the curved edges above and below.

```

public Path[] quadraticCurves(int size) {

    Path PATHS[] = new Path[size];

    /*
     * ABOVE CURVES
     */

    for (int i = 0; i < size; i = i + 2) {
        // draw quadratic path
        Path path = new Path();

        // Initializing MoveTo => specifying start coordinates
        MoveTo moveTo = new MoveTo();
        moveTo.setX((0 + 15) + (MainProgram.permutation.get(i) - 1) * 3 * 13);
        moveTo.setY(0);

        // Instantiating the class QuadCurveTo
        QuadCurveTo quadCurveTo = new QuadCurveTo();

        // Setting properties of the class QuadCurve
        try {
            quadCurveTo.setX((0 + 15) + ((MainProgram.permutation.get(i + 1) - 1) * 3 * 13));
        } catch (IndexOutOfBoundsException e) {
            quadCurveTo.setX((0 + 15) + ((MainProgram.permutation.get(0)) * 3 * 13));
        }
        quadCurveTo.setY(0);
        quadCurveTo.setControlX(0 + 15 + 5 * 3 * 13);
        quadCurveTo.setControlY(-140.0f + i * 11);

        // Adding the path elements to Observable list of the Path class
        path.getElements().add(moveTo);
        path.getElements().add(quadCurveTo);
        path.setStrokeWidth(1.5);
        // determining whether the path is going left or right
        try {
            // if the permutation[i] < permutation[i+1], then pathway is from LEFT to RIGHT

```

```

path.setStrokeWidth(1.5);
// determining whether the path is going left or right
try {
    // if the permutation[i] < permutation[i+1], then pathway is from LEFT to RIGHT
    if (MainProgram.permutation.get(i) < MainProgram.permutation.get(i + 1))
        path.setStroke(Color.DARKBLUE);
    // else RIGHT to LEFT
    else
        path.setStroke(Color.RED);
} catch (Exception e) {
    // if permutation[last_index] < permutation[0], then pathway is going to go from
    // LEFT to RIGHT
    if (MainProgram.permutation.get(0) > MainProgram.permutation.get(i))
        path.setStroke(Color.DARKBLUE);
    // else RIGHT to LEFT
    else
        path.setStroke(Color.RED);
}

PATHS[i] = path;
}

/*
 * BELOW CURVES
 */

for (int i = 1; i < size; i = i + 2) {
    // draw quadratic path
    Path path = new Path();

    // Initializing MoveTo => specifying start coordinates
    MoveTo moveTo = new MoveTo();
    moveTo.setX((0 + 15) + (MainProgram.permutation.get(i) - 1) * 3 * 13);
    moveTo.setY(0 + 30);

    // Instantiating the class QuadCurveTo
    QuadCurveTo quadCurveTo = new QuadCurveTo();

```

```

QuadCurveTo quadCurveTo = new QuadCurveTo();

// Setting properties of the class QuadCurve
try {
    quadCurveTo.setX((0 + 15) + ((MainProgram.permutation.get(i + 1) - 1) * 3 * 13));
} catch (IndexOutOfBoundsException e) {
    quadCurveTo.setX((0 + 15) + ((MainProgram.permutation.get(0) - 1) * 3 * 13));
}
quadCurveTo.setY(0 + 30);
try {
    quadCurveTo.setControlX(0 + 15
        + (MainProgram.permutation.get(i + 1) + MainProgram.permutation.get(i) - 1) / 2 * 3 * 13);
} catch (Exception e) {
    quadCurveTo.setControlX(
        0 + 15 + (MainProgram.permutation.get(0) + MainProgram.permutation.get(i) - 1) / 2 * 3 * 13);
}
quadCurveTo.setControlY(-140.0f + 140 + 30 + i * 14);

// Adding the path elements to Observable list of the Path class
path.getElements().add(moveTo);
path.getElements().add(quadCurveTo);
path.setStrokeWidth(1.5);

// determining whether the path is going left or right
try {
    // if the permutation[i] < permutation[i+1], then pathway is from LEFT to RIGHT
    if (MainProgram.permutation.get(i) < MainProgram.permutation.get(i + 1))
        path.setStroke(Color.DARKBLUE);
    // else RIGHT to LEFT
    else
        path.setStroke(Color.RED);
} catch (Exception e) {
    // if permutation[last_index] < permutation[0], then pathway is going to go from
    // LEFT to RIGHT
    if (MainProgram.permutation.get(0) > MainProgram.permutation.get(i))
        path.setStroke(Color.DARKBLUE);
    // else RIGHT to LEFT

```

```

        // else RIGHT to LEFT
        if (MainProgram.permutation.get(0) > MainProgram.permutation.get(i))
            path.setStroke(Color.DARKBLUE);
        // else RIGHT to LEFT
        else
            path.setStroke(Color.RED);
    }
    PATHS[i] = path;
}
return PATHS;
}

public boolean checkForIntersections() {
    Path setOfPaths[] = quadraticCurves(MainProgram.permutation.size());

    // checking the first half of paths (above)
    for (int i = 0; i < MainProgram.permutation.size(); i += 2)
        for (int j = 0; j < i; j += 2)
            if (setOfPaths[i].intersect(setOfPaths[j]) != null)
                return true;

    // checking below paths
    for (int i = 1; i < MainProgram.permutation.size(); i += 2)
        for (int j = 1; j < i; j += 2)
            if (setOfPaths[i].intersect(setOfPaths[j]) != null)
                return true;

    // if no intersection is present
    return false;
}
}

```


GRID TOUR

Grid tour can only exist for even-length permutations. So, this program assumes purely that an even-length permutation is generated. If an odd number is generated, then user won't be able to view a grid tour at all. He'll be prompted with a message, stating that he needs to generate maybe a new one.

The grid tour used a graph in form of DFS, but with a few constraints:

- Only right turns can exist (this is a relief because that means less search paths).
- Once a row/column is traversed and left alone, it cannot be revisited. (Another relieving constraint because this also limits the number of search paths available).

```
10 import java.util.ArrayList;
1
2 public class GridTours {
3
4     Pane pane;
5     static int sizeOfCellWidth = 27 - (2 * MainProgram.permutation.size()),
6           sizeOfCellHeight = 27 - (2 * MainProgram.permutation.size()), boardWidth = sizeOfCellWidth * 12,
7           boardHeight = sizeOfCellHeight * 12;
8     static int cellsX = boardWidth / sizeOfCellWidth;
9     static int cellsY = boardHeight / sizeOfCellHeight;
10    public static NxNGrid[][] grid = new NxNGrid[cellsX * 2 + 1][cellsY * 2 + 1];
11
12    // for 2nd permutation
13    int[] array = new int[MainProgram.permutation.size()];
14    ArrayList<Integer> rows = new ArrayList<Integer>(), columns = new ArrayList<Integer>();
15    ArrayList<Integer> visitedRows = new ArrayList<Integer>(), visitedColumns = new ArrayList<Integer>();
16    int currentRow = -1, currentColumn = -1, visitedRow = -1, visitedColumn = -1;
17
18    public void removePreviousRepresentation(int whichMenu) {
19        cellsX = MainProgram.permutation.size();
20        cellsY = MainProgram.permutation.size();
21        try {
22            // delete the previous representation (whatever it is)
23            switch (whichMenu) {
24                case 1:
25                    MainProgram.canvas1.getItems().remove(1);
26                    break;
27                case 2:
28                    MainProgram.canvas2.getItems().remove(1);
29                    break;
30                case 3:
31                    MainProgram.canvas3.getItems().remove(1);
32                    break;
33                case 4:
34                    MainProgram.canvas4.getItems().remove(1);
35                    break;
36            }
37        }
38    }
39 }
```

```

    }
} catch (Exception e) {
    // ignore

    /*
     * This exception is raised when there is initially nothing present (index out
     * of bounds exception)
     */
}
}

static class NxNGrid extends StackPane {

    // each rectangle is defined as a square.
    // A square = 1 cell;
    // A grid = (cellsX) * (cellsY) number of squares/cells
    Rectangle border = new Rectangle(sizeOfCellWidth, sizeOfCellHeight);

    // This constructor is called each time a new cell has to be defined
    public NxNGrid(int x, int y) {
        // Add each cell as the pane's children
        getChildren().add(border);
        // Each cell's dimensions are of sizeOfCellWidth and sizeOfCellHeight
        setTranslateX(x * sizeOfCellWidth);
        setTranslateY(y * sizeOfCellHeight);
        // Each cell is initially of a light grey color
        border.setFill(Color.LIGHTGREY);
        if (x % 2 == 1 && y % 2 == 1) // create alternate cells initially (not visible to the user)
            setColor1();
        else
            setColor2();
        // if(x==0 || x==cellsX*2+1)
        // setColor2();
        // if(x==0)
        // setColor2();
    }
}

```

```

2
3 public void setColor1() {
4     border.setFill(Color.DARKGREY);
5 }
6
7 public void setColor2() {
8     border.setFill(Color.BLUE);
9 }
10
11 }
12
13 public Pane createGrid() {
14     pane = new Pane();
15     pane.setPadding(new Insets(10));
16     // Define board's dimensions
17     sizeOfCellWidth = 32 - (2 * MainProgram.permutation.size());
18     sizeOfCellHeight = 32 - (2 * MainProgram.permutation.size());
19     pane.setPrefSize(boardWidth, boardHeight);
20
21     // Fill the board with cells
22     for (int i = 0; i <= cellsX * 2; i++) {
23         for (int j = 0; j <= cellsY * 2; j++) {
24             grid[i][j] = new NxNGrid(i, j);
25             pane.getChildren().add(grid[i][j]);
26         }
27     }
28
29     for (int i = 0; i < MainProgram.permutation.size(); i++) {
30         rows.add(i);
31         columns.add(i);
32     }
33
34     // call colorGrid() method
35     int count = 0;
36     int i = 0;
37     while (incomplete()) {
38

```

```

// call colorGrid() method
int count = 0;
int i = 0;
while (incomplete()) {
    count++;
    if (count >= 3 * MainProgram.permutation.size())
        break;
    i++;
    int matrixVariables[] = colorGrid(MainProgram.permutation.get(rows.get(rows.size() - 1)) - 1, array[i]);
    currentRow = matrixVariables[0];
    currentColumn = matrixVariables[1];

    int[] dummy = new int[array.length];
    boolean breakFlag = false;
    for (int j = 0; j < array.length; j++) {
        try {
            dummy[j] = array[j]; // see if NullPointerException occurs.
            // if so, then it means that some of the indices of array[] have been assigned a
            // value yet.
            // this means that 2nd permutation has not been fully generated yet.
        } catch (NullPointerException e) {
            if (j == array.length - 1)
                breakFlag = true; // no exception occurs=>2nd permutation generated
            continue;
        }
    }
    if (checkForRightTurns() && rows.size() == 0 && columns.size() == 0 && breakFlag)
        break;
    else {
        // restart the loop again if there are any left turns present
        i = 0;
        count = 0;
        for (int j = 0; j < MainProgram.permutation.size(); j++) {
            rows.add(j);
            columns.add(j);
        }
        continue;
    }
}

```

```

if (incomplete())
    colorGrid(rows.get(0), columns.get(0));

if (complete() && checkForRightTurns()) {
    redSquares(); // highlight the red colored squares.
}
Text helpText = new Text();

String originalPermutation = new String(" ");
for (int j = 0; j < MainProgram.permutation.size(); j++)
    originalPermutation += MainProgram.permutation.get(j) + " ";
originalPermutation += " ";
if (complete())
    helpText = new Text("Begin with a grid of N horizontal lines and N vertices lines.\n"
        + "Here, N is the length of permutation. Also, (N%2)=0.\n"
        + "The paths on the grid follow specific rules: each turn is\na right turn by 90 degrees.\n"
        + "The path has exactly 1 segment on each horizontal line and \n1 segment on each vertical line.\n"
        + "You need 2 permutations in order to draw a grid tour. \nOne represents the row and the other represents column."
        + "\nRED squares simply represent a turn.\nGREEN squares represent the pathway on that turn.\n\n"
        + "Original permutation:" + originalPermutation + "\n" + "Generated permutation:"
        + displaySecondPermutation(array));

// add all of the above information as children of a second pane and return this
// pane
Pane pane2 = new Pane();
HBox hbox = new HBox();

hbox.getChildren().addAll(pane, helpText);
pane2.getChildren().addAll(hbox);
// Return pane filled with children
return pane2;
}

private boolean incomplete() {
    if (columns.size() != 0 || rows.size() != 0) // if all rows & columns haven't been visited, return false
        return false;
}

```

```

private boolean incomplete() {
    if (columns.size() != 0 || rows.size() != 0) // if all rows & columns haven't been visited, return false
        return false;
    else
        return true;
}

private boolean complete() {
    if (columns.size() == 0 || rows.size() == 0) // if all rows & columns haven't been visited, return false
        return true;
    else
        return false;
}

public int[] colorGrid(int row, int column) {
    Random random = new Random();
    if (row == -1 && column == -1) {
        // pick a row && column to start with
        row = random.nextInt(MainProgram.permutation.size() - 1) * 2;
        column = random.nextInt(MainProgram.permutation.size() - 1) * 2;
        currentRow = row;
        currentColumn = column;
        grid[currentColumn][currentRow].border.setFill(Color.DARKBLUE);
        // add that row & column to visited and update current node
        visitedRow = -1;
        visitedColumn = -1;
        visitedRows.add(currentRow);
        visitedColumns.add(currentColumn);
        // remove that row and column from rows/columns to be traversed
        rows.remove(rows.get(currentRow));
        columns.remove(columns.get(currentColumn));
        // start with 2nd permutation
        array[0] = currentColumn;
    } else {
        visitedRow = currentRow;
    }
}

```

actions project > src > (default package) > GridTools >

```
visitedRow = currentRow;
visitedColumn = currentColumn;
currentRow = row;
currentColumn = column;
// don't visit traversed nodes again.
int newAddition = -1; // dummy initialization
for (int i = 0; i < rows.get(row); i++) {
    grid[i][columns.get(column)].border.setFill(Color.LIME);
}
for (int j = 0; j < columns.get(column); j++) {
    grid[rows.get(row)][j].border.setFill(Color.LIME);
    if (j == columns.get(column) - 1)
        newAddition = columns.get(column);
}
// add that to the new permutation.
int dummy[] = new int[array.length];
boolean breakFlag = false;
int additionIndex = -1;
for (int j = 0; j < array.length; j++) {
    try {
        dummy[j] = array[j]; // see if NullPointerException occurs.
        // if so, then it means that some of the indices of array[] have been assigned a
        // value yet.
        // this means that 2nd permutation has not been fully generated yet.
        // wherever the FIRST exception occurs, that's the next addition of the 2nd
        // permutation
        additionIndex = j;
    } catch (NullPointerException e) {
        breakFlag = true;
        break;
    }
    if (breakFlag)
        break;
}
array[additionIndex] = columns.get(newAddition);
}
return array;
```

```

    }

    public boolean checkForRightTurns() {
        for (int i = 0; i < cellsX * 2 + 1; i++) {
            for (int j = 0; j < cellsY * 2 + 1; j++) {
                // only apply this method if the grid has been colored with anything other than
                // default grey
                if (grid[i][j].border.getFill().equals(Color.DARKBLUE) || grid[i][j].border.getFill().equals(Color.LIME)
                    || grid[i][j].border.getFill().equals(Color.RED)) {

                    // check for each valid turn.

                    // a valid turn occurs when grid[i][j]'s color matches one of its diagonal
                    // neighbor's colors
                    if (i > j) {
                        if (!grid[2 * i][2 * j + 1].border.getFill().equals(Color.LIME))
                            return false;
                    }
                    if (i == j) {
                        if (!grid[2 * i - 1][2 * j].border.getFill().equals(Color.LIME))
                            return false;
                    }
                    if (i < j) {
                        if (!grid[2 * i][2 * j - 1].border.getFill().equals(Color.LIME))
                            return false;
                    }
                } else// ignore
                    continue;
            }
        }
        return true;
    }

    public void redSquares() {

```

```

    }

    public void redSquares() {
        for (int i = 0; i < cellsX * 2 + 1; i++) {
            for (int j = 0; j < cellsY * 2 + 1; j++) {
                // only apply this method if the grid has been colored with anything other than
                // default grey
                if (grid[i][j].border.getFill().equals(Color.DARKBLUE)
                    || grid[i][j].border.getFill().equals(Color.LIME)) {
                    // check each diagonal neighbor
                    // 4 different scenarios can exist!

                    // scenario one: if one green/blue square is [row][column] & the other is at
                    // [row+1][column+1]
                    if ((grid[i][j].border.getFill().equals(Color.LIME)
                        || grid[i][j].border.getFill().equals(Color.DARKBLUE))
                        && (grid[i + 1][j + 1].border.getFill().equals(Color.LIME)
                            || grid[i + 1][j + 1].border.getFill().equals(Color.DARKBLUE))) {
                        if ((grid[i - 1][j].border.getFill().equals(Color.LIME)
                            || grid[i - 1][j].border.getFill().equals(Color.DARKBLUE))
                            && (grid[i + 1][j + 1].border.getFill().equals(Color.LIME)
                                || grid[i + 1][j + 1].border.getFill().equals(Color.DARKBLUE)))
                            grid[i + 1][j].border.setFill(Color.RED);
                    }

                    // scenario two: if one green/blue square is [row+1][column] & the other is at
                    // [row][column+1]
                    if ((grid[i + 1][j].border.getFill().equals(Color.LIME)
                        || grid[i + 1][j].border.getFill().equals(Color.DARKBLUE))
                        && (grid[i][j + 1].border.getFill().equals(Color.LIME)
                            || grid[i][j + 1].border.getFill().equals(Color.DARKBLUE))) {
                        if ((grid[i + 1 + 1][j].border.getFill().equals(Color.LIME)
                            || grid[i + 1 + 1][j].border.getFill().equals(Color.DARKBLUE))
                            && (grid[i + 1 - 1][j + 1 + 1].border.getFill().equals(Color.LIME)
                                || grid[i + 1 - 1][j + 1 + 1].border.getFill().equals(Color.DARKBLUE)))
                            grid[i][j].border.setFill(Color.RED);
                    }
                }
            }
        }
    }

```

ermutationsProject ▶ src ▶ (default package) ▶ Gridours ▶

```
// [row+1][column+1] (symmetric to scenario 1)
if ((grid[i][j].border.getFill().equals(Color.LIME)
    || grid[i][j].border.getFill().equals(Color.DARKBLUE))
    && (grid[i + 1][j + 1].border.getFill().equals(Color.LIME)
    || grid[i + 1][j + 1].border.getFill().equals(Color.DARKBLUE))) {
    if ((grid[i][j - 1].border.getFill().equals(Color.LIME)
        || grid[i][j - 1].border.getFill().equals(Color.DARKBLUE))
        && (grid[i + 1][j + 1].border.getFill().equals(Color.LIME)
        || grid[i + 1][j + 1].border.getFill().equals(Color.DARKBLUE)))
        grid[i][j + 1].border.setFill(Color.RED);
}

// scenario four: if one green/blue square is [row+1][column] & the other is at
// [row][column+1] (symmetric to scenario two)
if ((grid[i + 1][j].border.getFill().equals(Color.LIME)
    || grid[i + 1][j].border.getFill().equals(Color.DARKBLUE))
    && (grid[i][j + 1].border.getFill().equals(Color.LIME)
    || grid[i][j + 1].border.getFill().equals(Color.DARKBLUE))) {
    if ((grid[i + 1][j - 1].border.getFill().equals(Color.LIME)
        || grid[i + 1][j - 1].border.getFill().equals(Color.DARKBLUE))
        && (grid[i - 1][j + 1].border.getFill().equals(Color.LIME)
        || grid[i - 1][j + 1].border.getFill().equals(Color.DARKBLUE)))
        grid[i][j].border.setFill(Color.RED);
}
}
else
    continue;// ignore
}
}

public String displaySecondPermutation(int[] array) {
    String string = new String();
    string += "(";
```

```

}

public String displaySecondPermutation(int[] array) {
    String string = new String();
    string += "(";
    for (int i = 0; i < array.length; i++) {
        string += Integer.toString(array[i]);
    }
    string += " )";
    return string;
}
}
```


REJECTED DATA STRUCTURES

LINKED LISTS

This data structure was not only rejected because using it was against the rules of this project, but it was also rejected because using links would lead to unnecessary time complexity. Not only that, but there will serious cases of dangling pointers within the Java's heap memory. Therefore, without much thought, linked lists were not even considered.

CACTUS STACK

This data structure was first considered for implementing a grid tour because rather than keeping track of children like most graphs do, this data structure lets the children keep track of their parents. But going into the later stages of programming for grid tours, the only way that I could come up with to keep track of the parents is to use some sort of *link* and pointers (these are abstracted by Java, but nonetheless, they do exist).

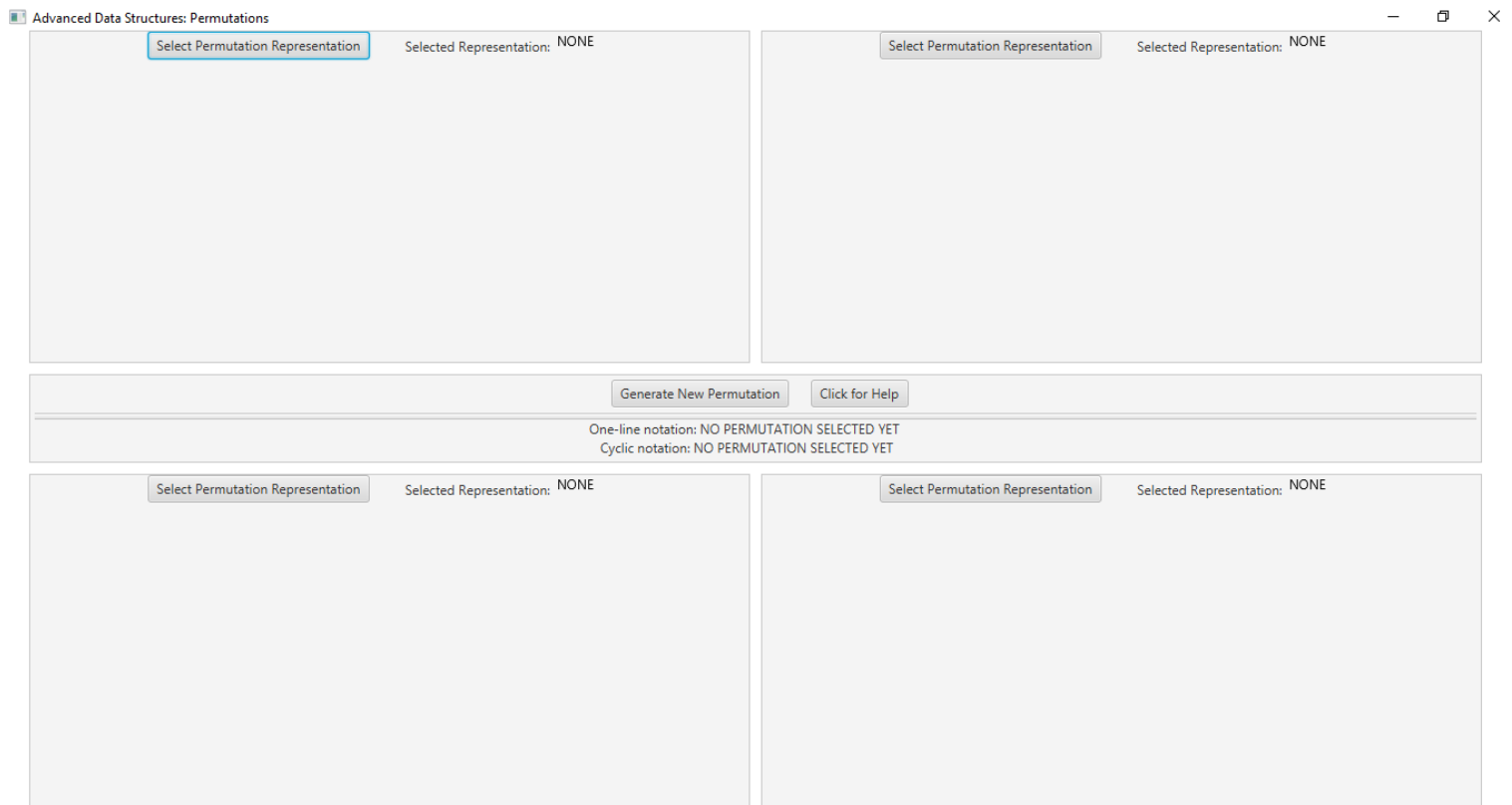
A cactus stack using arrays is not possible because if the user selects a large permutation size, then the program would run considerably slow because there are so many paths to check for in a grid tour. Therefore, links have to come into the field somehow.

This not only violates the project's rules, but it also makes a mess out of heap memory. Time, space, and performance are affected and not in a good way. Therefore, this data structure was rejected.

SCREEN VISUALS (from user's perspective)

The data structures for the screen visuals was more about placing how the data is organized. Therefore, it was important to understand how mouse listeners worked and to know which mouse event is activated via a user's actions.

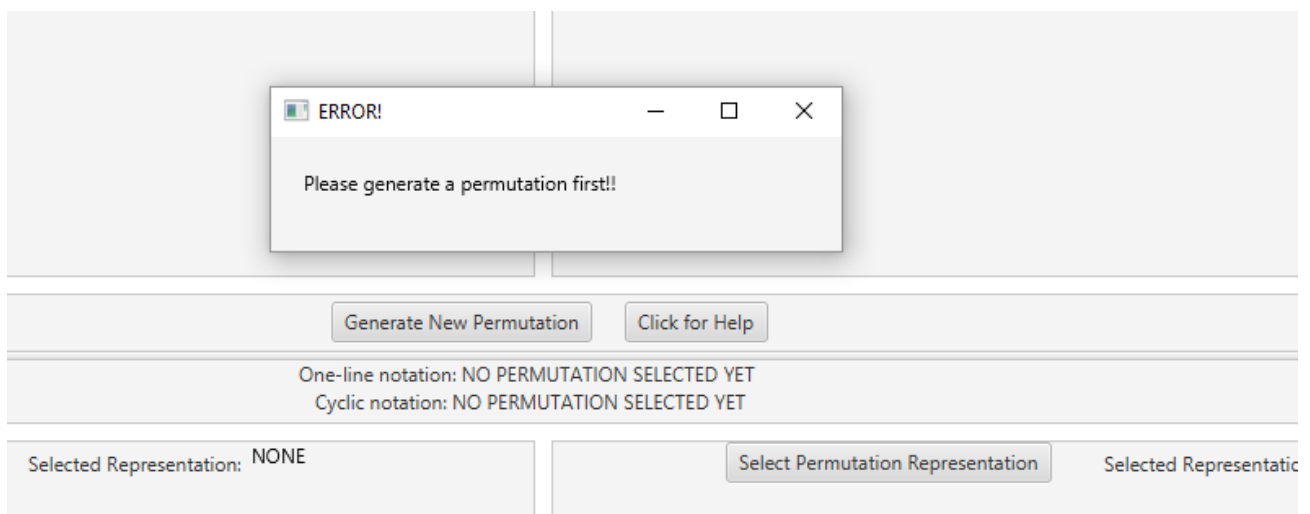
First, upon starting the application, the user will be met with the following visual:



From the above screenshot, there are five total partitions of the entire screen. Four of them are for displaying permutation representations while the other is to either seek help regarding the representation *or* generate the permutation itself. Each partition is portrayed through a **Pane**.

Next, user should generate the permutation. The user won't be able to access 'Select Permutation Representation' on either of the four panes.

If he does so, an error dialog box will be displayed.



Therefore, generating a permutation is necessary. The button for that is in the center of the screen.

Generating New Permutation...

Enter your own permutation (separated by spaces):

Enter the size of random permutation:

Note the text in grey color. This should serve as a hint for the user as to how to enter the permutation.

The user has two options to choose from:

1. Input the permutation himself, *provided*:
 - a. A permutation of length N must include the digits from 1 to N , inclusive.
 - b. The length of the permutation must be greater than 1.
 - c. The length of the permutation cannot exceed by 12.
 - d. Each digit should contain at least one space character (otherwise, without spacing, consecutive digits are considered to be one number). The reason for this is because since the maximum allowed length is 12, a conflict of ambiguity, for example, between 1 and 11 will result.
2. Generate a random permutation of length N (this is provided by user), *provided*:
 - a. The inputted length is between 2 and 12 (inclusive).

If permutation is wrongly entered:

Generating New Permutation...

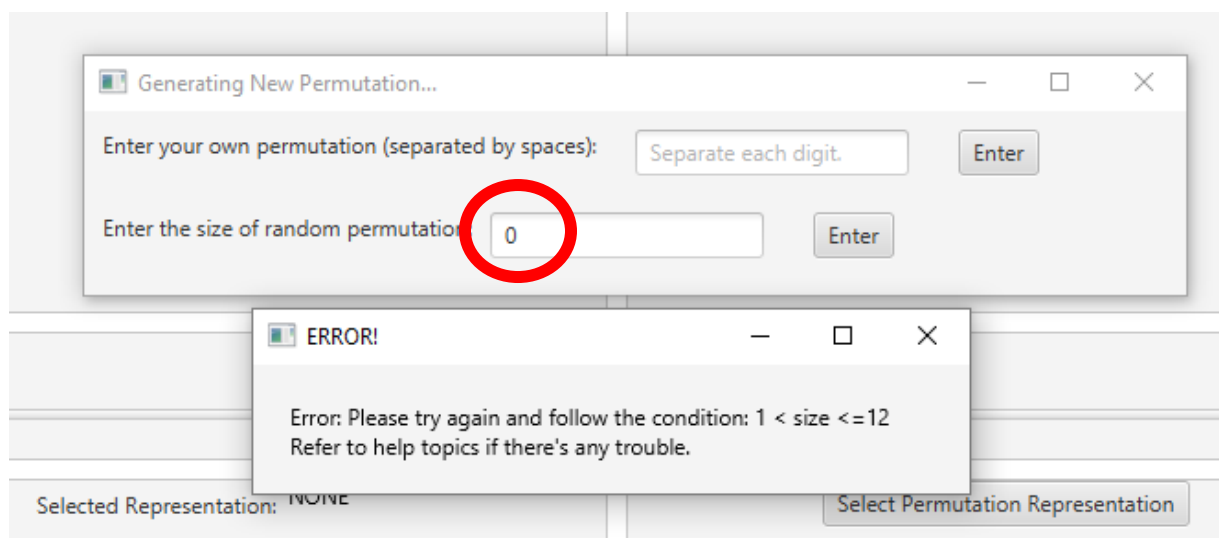
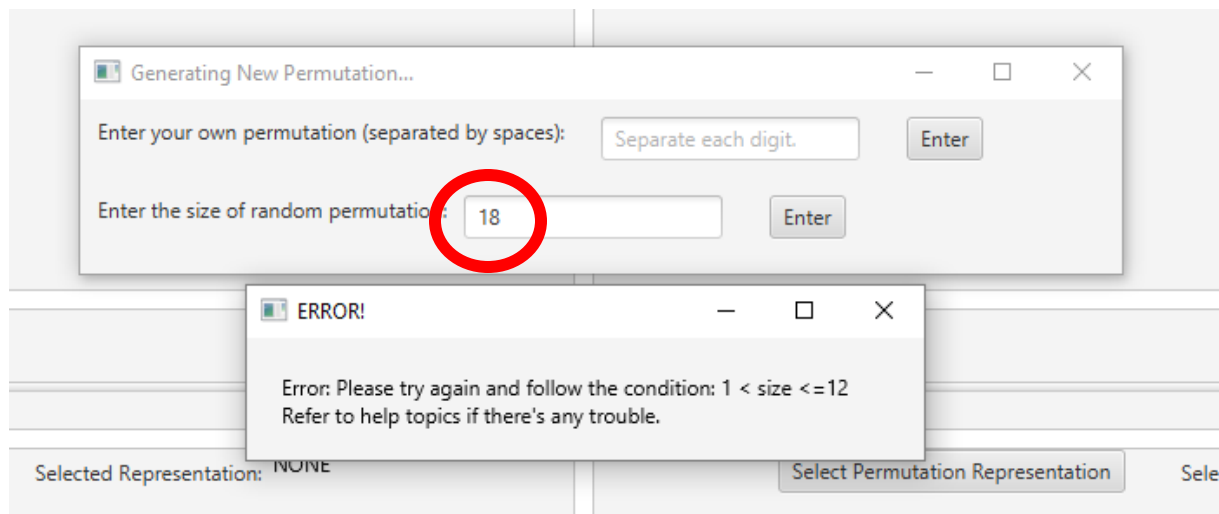
Enter your own permutation (separated by spaces):

Enter the size of random permutation:

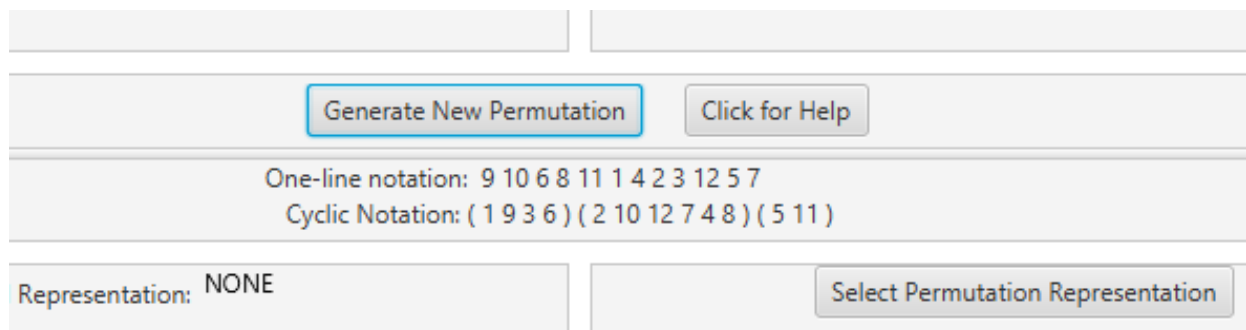
ERROR!

Permutation of size N should include distinct numbers from 1 to N.

If length for random permutation generation is not between 2 and 12:



After a permutation is validated: (the below screenshot is a random permutation)



AT ANY TIME, user is able to access help.

Generate New Permutation

Click for Help

One-line notation: NO PERMUTATION SELECTED YET

Cyclic notation: NO PERMUTATION SELECTED YET

Selected Representation: NONE

Select Permutation Represent

Permutations and Representations

HOW TO USE THE GUI

You'll see 4 partitions in the screen. Each partition can hold only one representation.
Click on 'Generate Permutation' to get started.
You have two options here: enter the permutation or generate one randomly based on a fixed length
If you choose to enter a permutation, make sure that the numbers are separated by AT LEAST a single space.
Also, the length of permutation must at least be 2 and less than 13.
If choose to generate a random permutation, enter an appropriate number ONLY between 2 and 12 (inclusive).
Click on 'Enter' and you'll see the permutation in both one-line and cyclic notations.
Now, in any of the four partitions, you can choose to generate whichever representation you want.

SQUARE GRID REPRESENTATION

The square grid is one of the representations for a permutation.
The rules are that for a permutation of size N, a grid of NxN dimensions must be created first.
Column j of row i is colored RED whenever $\text{permutation}[i]=j$.
Then, GREEN colored squares represent inversions:
These are colored by seeing those squares which have colored squares lying both below (in same column) and to their right (same row).
If there are no GREEN squares, then the permutation is ordered in ascending order.

INVERSE

A permutation's inverse is another permutation of the same length with the same number of elements.
To find the inverse, first list out two rows, with one top lining up exactly on top of another.
The top row should be sorted, whereas the bottom resembles the original permutation.
Interchange the two rows and sort top new row in increasing order using the second one.
For example, say the first digit of top row is 10. And the first digit of bottom row is 5.
In this case, the digit 10 would be in the 5th position of the inverse.

INVERSION TABLE

Next, the user can select from any of the seven representations in each partition, where the representations are accessed through a menu button explained in one of the previous sections.

Menu Options

Square Grid with Inversions

Permutation's Inverse

Inversion Table

Factoradic number

Permutation Graph

Meander

Grid Tour

This menu option can only be selected if a valid permutation has been generated by either the user or the program.

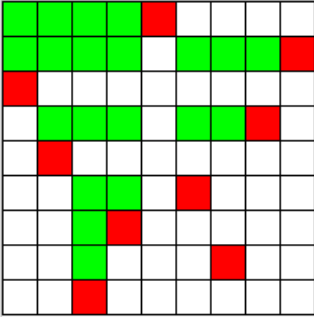
There is a total of seven representations available for the user:

1. Square grid with colored squares and inversions

Advanced Data Structures: Permutations

Select Permutation Representation

Selected Representation: Square Grid with Inversions



The square grid on the left shows one of the representations for a permutation.

The rules are that for a permutation of size N , a grid of $N \times N$ dimensions must be created first.

Column j of row i is colored RED whenever $\text{permutation}[i] = j$.

Then, GREEN colored squares represent inversions: These are colored by seeing those squares which have colored squares lying both below (in same column) and to their right (same row). If there are no GREEN squares, then the permutation is ordered.

Generate New Permutation

Click

One-line notation: 5 9 1 8 2 6 4 7 3

Cyclic Notation: (1 5 2 9 3) (4 8 7) (6)

2. Inverse

Advanced Data Structures: Permutations

Select Permutation Representation

Selected Representation: Permutation's Inverse

INVERSE: 3 5 9 7 1 6 8 4 2

A permutation's inverse is another permutation of the same length with the same number of elements.

To find the inverse, first list out two rows, with one top lining up exactly on top of another. The top row should be sorted, whereas the bottom resembles the original permutation. Interchange the two rows and sort top new row in increasing order using the second one.

For example, say the first digit of top row is 10. And the first digit of bottom row is 5. In this case, the digit 10 would be in the 5th position of the inverse.

Generate New Permutation

Click

One-line notation: 5 9 1 8 2 6 4 7 3

Cyclic Notation: (1 5 2 9 3) (4 8 7) (6)

3. Inversion table

Select Permutation Representation

Selected Representation: Inversion Table

INVERSION TABLE: 2 3 6 4 0 2 2 1 0

An inversion table uniquely determines the corresponding permutation.

Inversion table $b_1, b_2, b_3, \dots, b_N$ of the permutation $a_1, a_2, a_3, \dots, a_N$ is obtained by letting b_j be the number of elements to the left of $a_i = j$ greater than j .
(i.e., b_j = number of inversions whose second component equals j .)

Generate New Permutation

Click f

One-line notation: 5 9 1 8 2 6 4 7 3

Cyclic Notation: (1 5 2 9 3) (4 8 7) (6)

4. Factoradic Number

Select Permutation Representation

Selected Representation: Factoradic Number

FACTORADIC REPRESENTATION: 100577

A factoradic number is a sequence of digits where each digit has a different radix in particular, the radix for digit $d_l = (n-l)!$, where l = index and N = length of permutation.
i.e., rightmost digit has base 1, next digit has base 2!, and so on such that each digit in a factoradic number is smaller than its base.

First, find the inversion table of the given permutation: 2 3 6 4 0 2 2 1 0
Next, calculate the factoradic number:

$$(2 \times 7!) + (3 \times 6!) + (6 \times 5!) + (4 \times 4!) + (0 \times 3!) + (2 \times 2!) + (2 \times 1!) + (1 \times 0!) = 100577$$

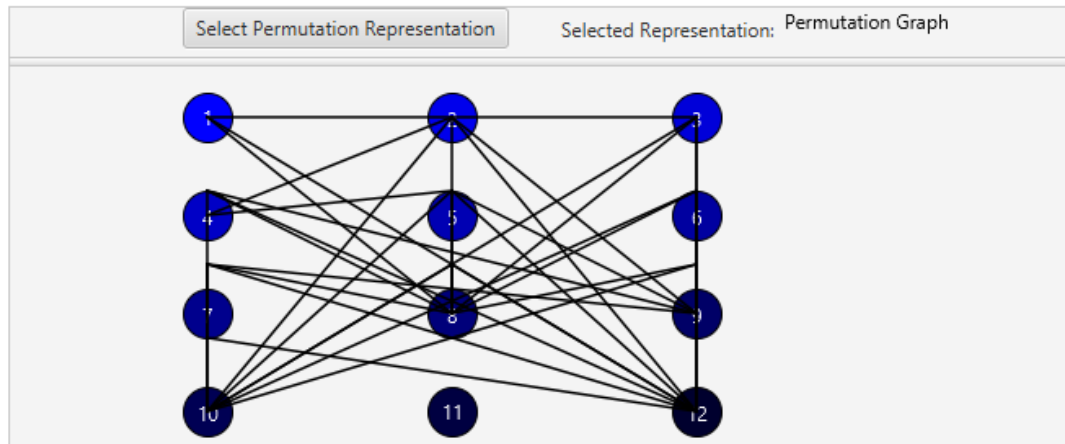
Generate New Permutation

Click f

One-line notation: 5 9 1 8 2 6 4 7 3

Cyclic Notation: (1 5 2 9 3) (4 8 7) (6)

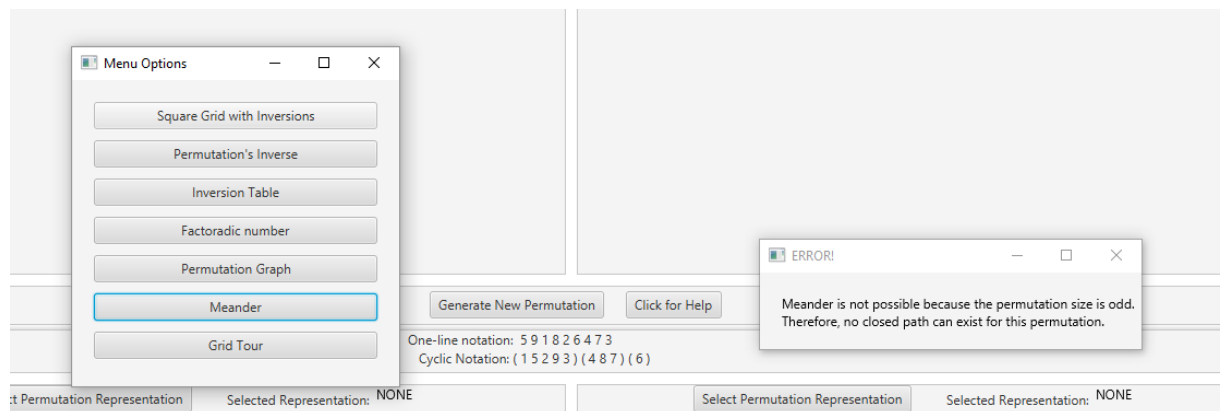
5. Permutation Graph



In the above picture, 11 is its correct place and hence, there are no edges from or to it.

6. Meander

Since the selected permutation from above is an odd number, a meander does not exist because there will be no closed loop.

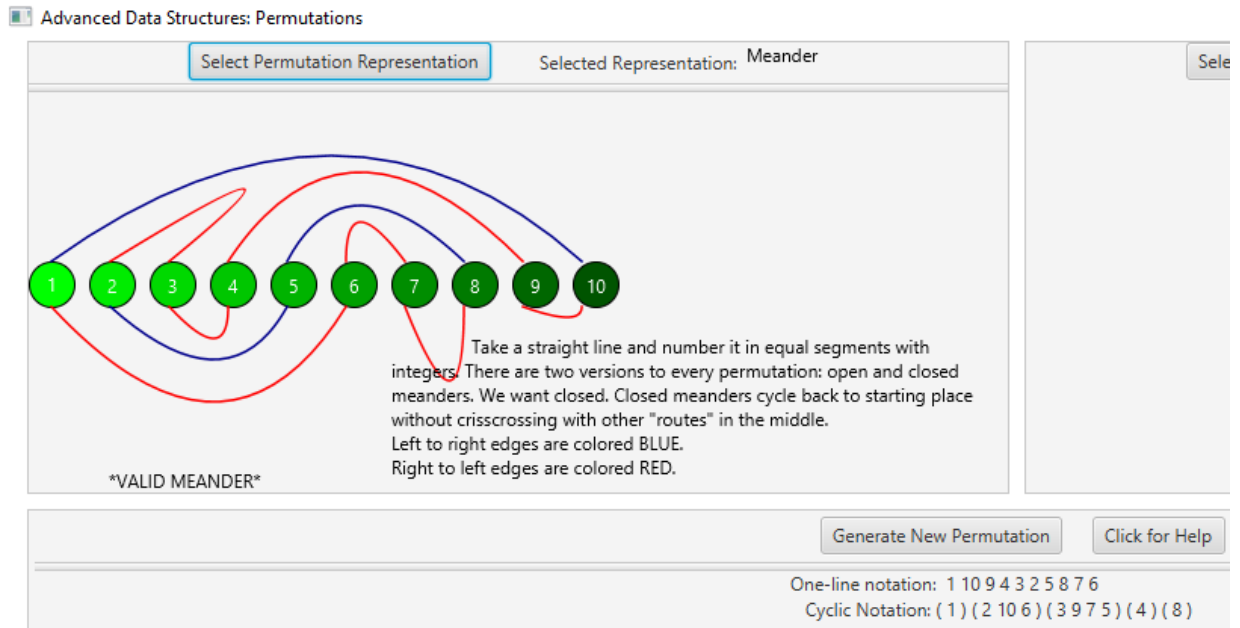


The above error message is only showed if the permutation size/length is odd. Also, I have merely dragged the windows around the stage, so that the content is visible. Normally, they appear in the center of the screen.

For a closed loop, a meander can certainly exist (provided that the above edges don't crisscross with each other and same with the below edges).

If the edges don't crisscross, it's a valid meander. Else, it's an invalid meander.

An example of a valid meander is shown below:

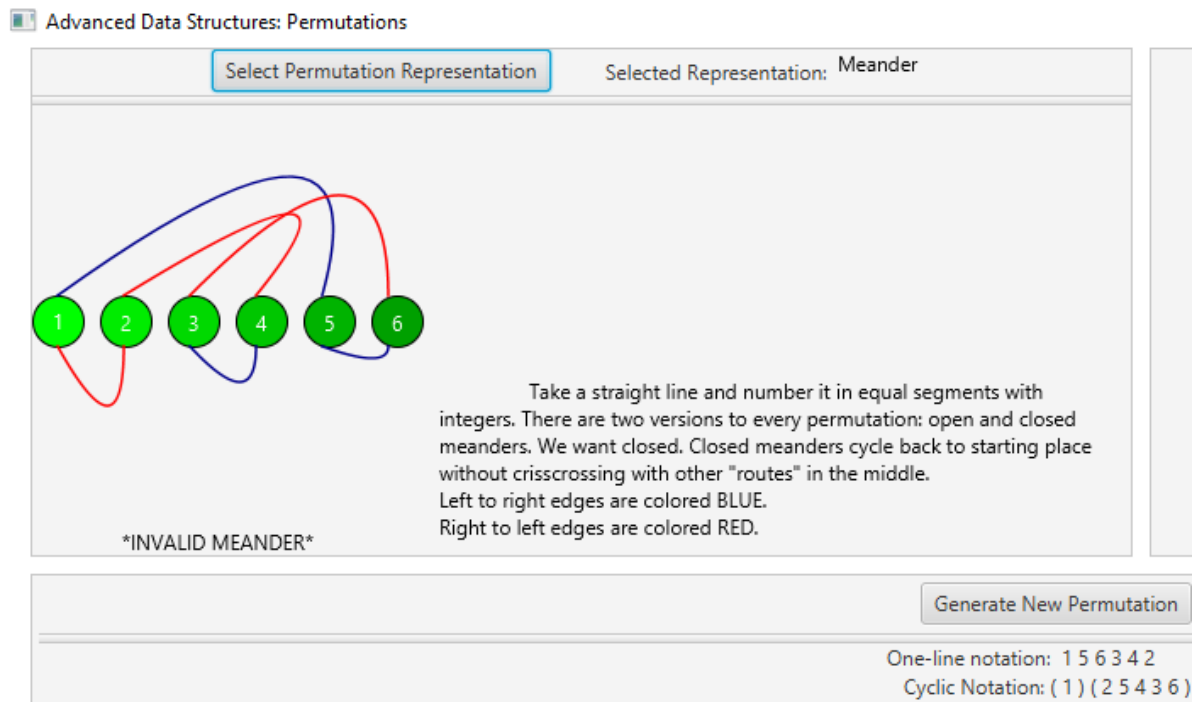


The edges that go from left to right are colored *blue* and the edges that go from right to left are colored *red*.

If the permutation size is *EVEN*, but edges crisscross anyway, then that meander is shown as well.

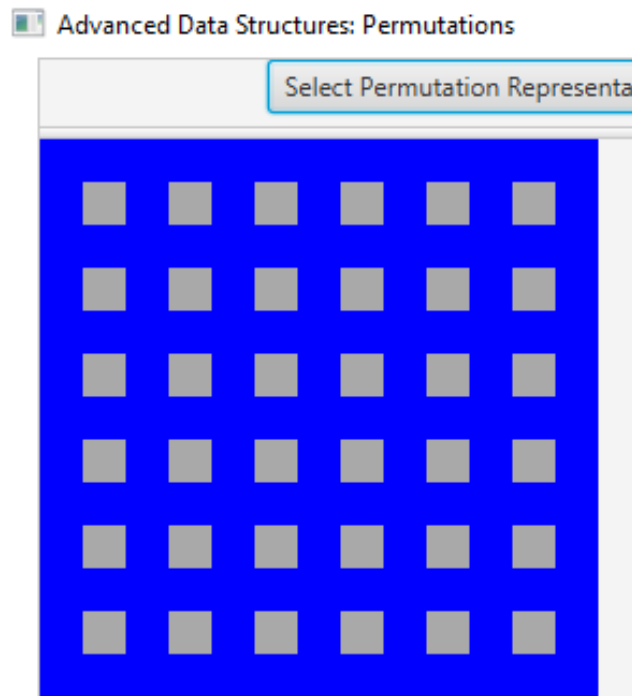
Occasionally, lines may intersect with the text because there wasn't enough space to include enough information.

An example of an invalid meander is shown below:



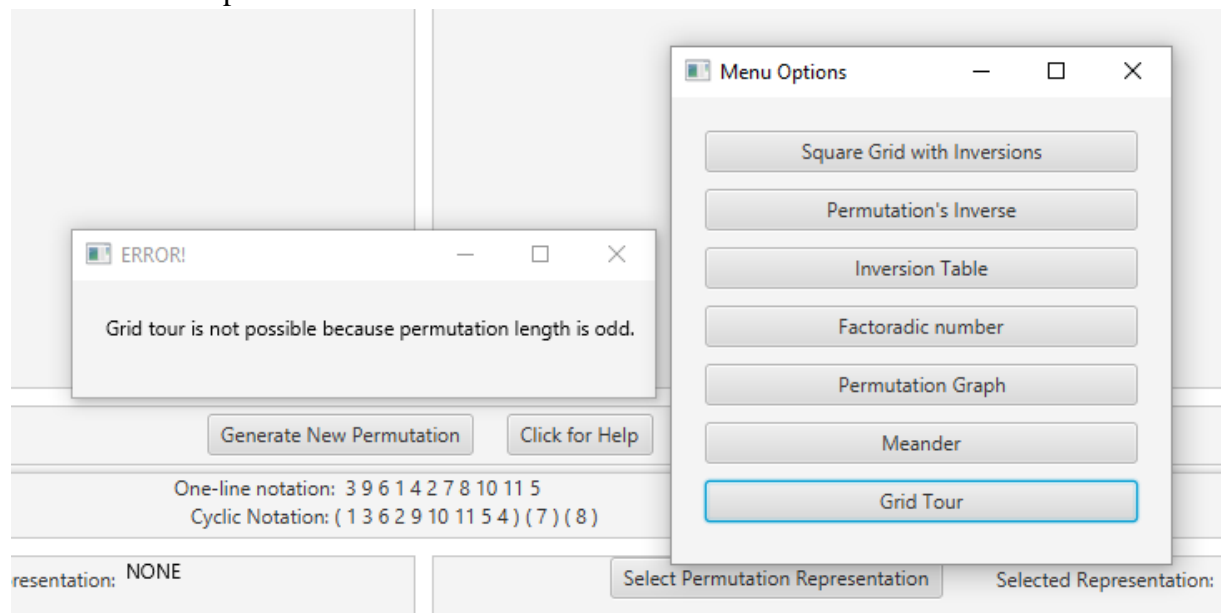
7. Grid Tour

As a side note, an initially developing grid tour will look like this:

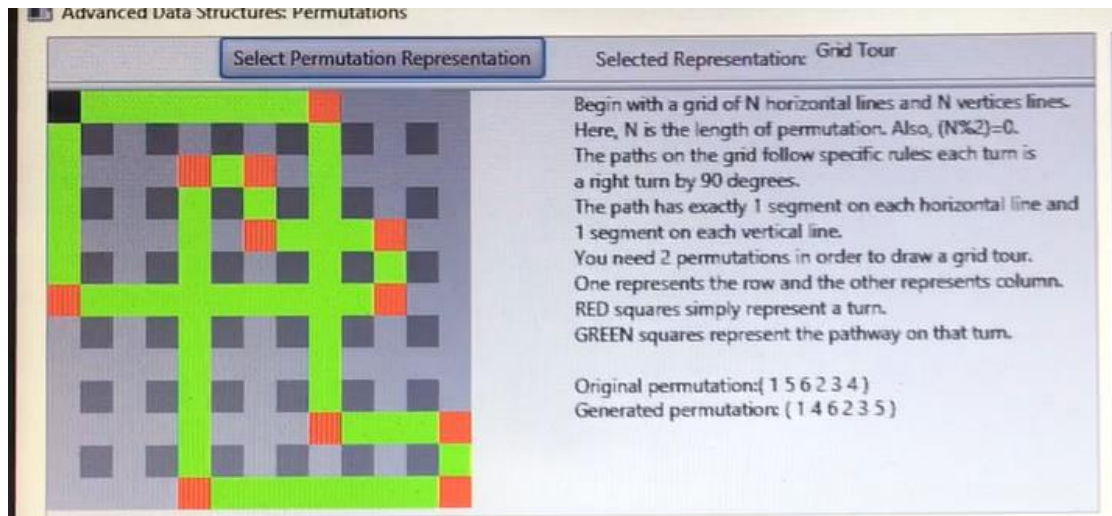


This is *NOT* shown to the user at all. It's only to start developing the code and to illustrate how that happens.

Moving on, like the meander, a grid tour is *not* possible if the length of permutation is odd. One of the examples is shown below:



If the permutation's length is even:



An example of how the overall screen may look like:

(A random permutation of size 12 is generated in the following picture)

In case, the center box is too small to read:

- Generated random permutation (one-line notation) of size 12 – **12 3 1 10 5 8 11 9 4 2 6 7**.
- Cyclic notation – **(1 12 7 11 6 8 9 4 10 2 3) (5)**

Advanced Data Structures: Permutations

Select Permutation Representation

Selected Representation: Square Grid with Inversions

The square grid on the left shows one of the representations for a permutation.

The rules are that for a permutation of size N, a grid of NxN dimensions must be created first.

Column j of row i is colored RED whenever permutation[i]=j.

Then, GREEN colored squares represent inversions: These are colored by seeing those squares which have colored squares lying both below (in same column) and to their right (same row). If there are no GREEN squares, then the permutation is ordered.

Select Permutation Representation

Selected Representation: Permutation's Inverse

INVERSE: 3 10 2 9 5 11 12 6 8 4 7 1

A permutation's inverse is another permutation of the same length with the same number of elements.

To find the inverse, first list out two rows, with one top lining up exactly on top of another. The top row should be sorted, whereas the bottom resembles the original permutation. Interchange the two rows and sort top new row in increasing order using the second one.

For example, say the first digit of top row is 10. And the first digit of bottom row is 5. In this case, the digit 10 would be in the 5th position of the inverse.

Generate New Permutation

Click for Help

One-line notation: 12 3 1 10 5 8 11 9 4 2 6 7

Cyclic Notation: (1 12 7 11 6 8 9 4 10 2 3) (5)

Select Permutation Representation

Selected Representation: Inversion Table

INVERSION TABLE: 2 8 1 6 2 5 5 2 3 1 1 0

An inversion table uniquely determines the corresponding permutation.

Inversion table b1, b2, b3, ..., bN of the permutation a1, a2, a3, ..., aN is obtained by letting bj be the number of elements to the left of ai = j greater than j.
(i.e., bj = number of inversions whose second component equals j)

Select Permutation Representation

Selected Representation: Factoradic Number

FACTORADIC REPRESENTATION: 109483149

A factoradic number is a sequence of digits where each digit has a different radix in particular, the radix for digit dl = (n-l), where l = index and N = length of permutation.

i.e., rightmost digit has base 1, next digit has base 2, and so on such that each digit in a factoradic number is smaller than its base.

First, find the inversion table of the given permutation: 2 8 1 6 2 5 5 2 3 1 1 0

Next, calculate the factoradic number:
 $(2 \times 10!) + (8 \times 9!) + (1 \times 8!) + (6 \times 7!) + (2 \times 6!) + (5 \times 5!) + (5 \times 4!) + (2 \times 3!) + (3 \times 2!) + (1 \times 1!) + (1 \times 0!) = 109483149$

METHODS OF DEBUGGING

EXCEPTIONAL HANDLING (via *try-catch {}* blocks)

1. Even though no linked lists were used whatsoever, there were plenty of exceptions in this category: *NullPointerException*. This exception occurred a couple times whenever an array's index is accessed. An array is usually declared to be of the same size as the permutation itself. But, whenever performing a few operations, if that array's particular index has not been initialized yet, this exception would be raised.
2. An *ArrayIndexOutOfBoundsException*, *IndexOutOfBoundsException* exceptions occur, for example, when the program tries to reset the four partitions free of any representations. The only time that they're reset is when a new permutation is generated. Each pane has two entries. One consists of an HBox with the menu options and labels for selected representation. And the other is the corresponding representation itself. These two entries are shown via a VBox. The last entry of the box is deleted (which means the index is 1.) If there is no representation, then these two exceptions will occur. To deal with that, the code is enclosed within *try-catch {}* blocks.

```
public static void resetPreviousRepresentations() {  
    // whenever a new permutation is generated, delete previous representations (if  
    // there is one available for each partition)  
  
    // check each partition independently of others  
    try {  
        canvas1.getItems().remove(1);  
    } catch (ArrayIndexOutOfBoundsException e) {  
        // ignore  
    } catch (IndexOutOfBoundsException e) {  
        // ignore  
    }  
  
    try {  
        canvas2.getItems().remove(1);  
    } catch (ArrayIndexOutOfBoundsException e) {  
        // ignore  
    } catch (IndexOutOfBoundsException e) {  
        // ignore  
    }  
  
    try {  
        canvas3.getItems().remove(1);  
    } catch (ArrayIndexOutOfBoundsException e) {  
        // ignore  
    } catch (IndexOutOfBoundsException e) {  
        // ignore  
    }  
  
    try {  
        canvas4.getItems().remove(1);  
    } catch (ArrayIndexOutOfBoundsException e) {  
        // ignore  
    } catch (IndexOutOfBoundsException e) {  
        // ignore  
    }  
}
```

REFERENCES

1. Oracle's Documentation of JavaFX
2. Class notes – many of the examples in this documentation are from the ones given in class (especially the ones given in screen visuals).
3. Google Images – most of the images used to define terminology are from Oracle (but found in Google). The specific references are listed below the image itself.