HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

OBJECT ORIENTED PROGRAMMING

MINI PROJECT REPORT

SORTING ALGORITHM VISUALIZATION

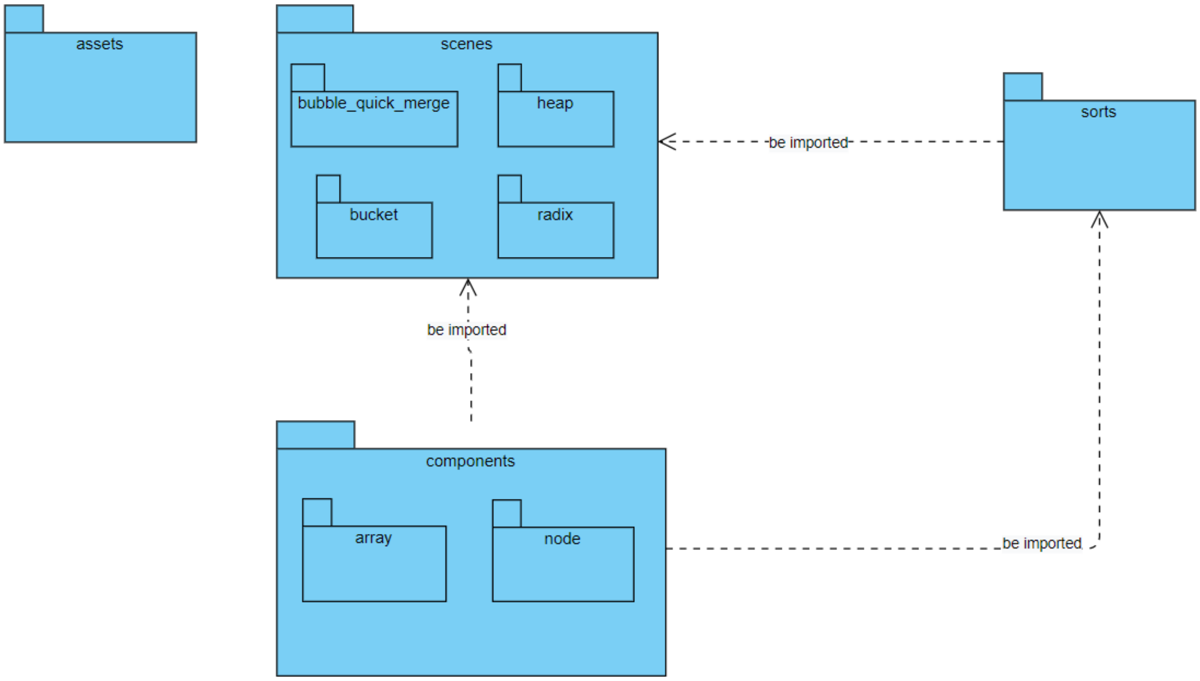
Instructor: Bui Thi Mai Anh

Student Name	**Roles**	**Student ID**	**Tasks**
Phan Xuân Tân	MEMBER	20194833	Radix Sort logic, Bucket Sort logic
Đặng Yến Trang	MEMBER	20190114	Main Menu, Bubble Sort, Quick Sort
Nguyễn Lưu Hoàng Minh	MEMBER	20194798	Heap Sort, Radix Sort animation, Bucket Sort animation
Lê Nguyễn Tuấn Minh	MEMBER	20194797	Merge Sort, Diagrams, Report

Hanoi, 2021

I. Classes & Packages

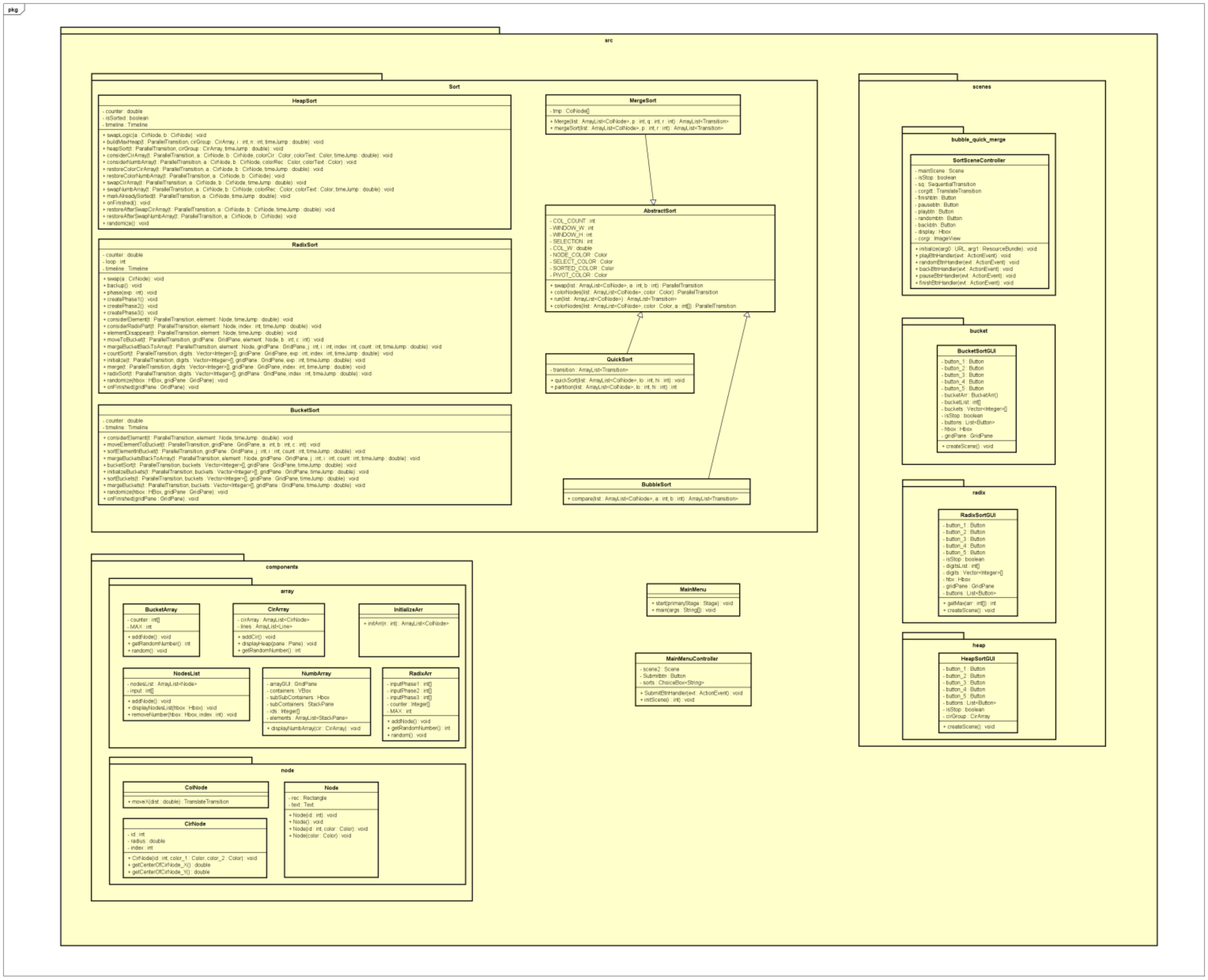
The package diagram:



The program consists of 4 **packages**:

1. The package "**assets**": containing images used in the user interface
2. The package "**components**": containing two sub-packages "*array*" and "*node*", all the column, circle, number, etc objects that would be used later for sorting
3. The sorting package "**sorts**": containing all the different algorithms for sorting. This package imports the "*components*" package, to initialize arrays or arraylists of objects in the "*components*" package.
4. The scene package "**scenes**": containing four sub-packages "bubble_quick_merge", "heap", "bucket", and "radix", has different scenes for different sorting algorithms. This package imports two packages "*components*" and "*sorts*".

The class diagram:


pkg
src
Sort
HeapSort
- counter : double
- isSorted : boolean
- timeline : Timeline
+ swapLogic(a : CirNode, b : CirNode) : void
+ buildMaxHeap(t : ParallelTransition, cirGroup : CirArray, i : int, n : int, timeJump : double) : void
+ heapSort(t : ParallelTransition, cirGroup : CirArray, timeJump : double) : void
+ considerCirArray(t : ParallelTransition, a : CirNode, b : CirNode, colorCir : Color, colorText : Color, timeJump : double) : void
+ considerNumbArray(t : ParallelTransition, a : CirNode, b : CirNode, colorRec : Color, colorText : Color) : void
+ restoreColorCirArray(t : ParallelTransition, a : CirNode, b : CirNode, timeJump : double) : void
+ restoreColorNumbArray(t : ParallelTransition, a : CirNode, b : CirNode) : void
+ swapCirArray(t : ParallelTransition, a : CirNode, b : CirNode, timeJump : double) : void
+ swapNumbArray(t : ParallelTransition, a : CirNode, b : CirNode, colorRec : Color, colorText : Color, timeJump : double) : void
+ markAlreadySorted(t : ParallelTransition, a : CirNode, timeJump : double) : void
+ onFinished() : void
+ restoreAfterSwapCirArray(t : ParallelTransition, a : CirNode, b : CirNode, timeJump : double) : void
+ restoreAfterSwapNumbArray(t : ParallelTransition, a : CirNode, b : CirNode) : void
+ randomize() : void
RadixSort
- counter : double
- loop : int
- timeline : Timeline
+ swap(a : CirNode) : void
+ backup() : void
+ phase(exp : int) : void
+ createPhase1() : void
+ createPhase2() : void
+ createPhase3() : void
+ considerElement(t : ParallelTransition, element : Node, timeJump : double) : void
+ considerRadixPart(t : ParallelTransition, element : Node, index : int, timeJump : double) : void
+ elementDisappear(t : ParallelTransition, element : Node, timeJump : double) : void
+ moveToBucket(t : ParallelTransition, gridPane : GridPane, element : Node, b : int, c : int) : void
+ mergeBucketBackToArray(t : ParallelTransition, element : Node, gridPane : GridPane, j : int, i : int, index : int, count : int, timeJump : double) : void
+ countSort(t : ParallelTransition, digits : Vector<Integer>[], gridPane : GridPane, exp : int, index : int, timeJump : double) : void
+ initialize(t : ParallelTransition, digits : Vector<Integer>[], gridPane : GridPane, exp : int, timeJump : double) : void
+ merge(t : ParallelTransition, digits : Vector<Integer>[], gridPane : GridPane, index : int, timeJump : double) : void
+ radixSort(t : ParallelTransition, digits : Vector<Integer>[], gridPane : GridPane, index : int, timeJump : double) : void
+ randomize(hbox : HBox, gridPane : GridPane) : void
+ onFinished(gridPane : GridPane) : void
BucketSort
- counter : double
- timeline : Timeline
+ considerElement(t : ParallelTransition, element : Node, timeJump : double) : void
+ moveElementToBucket(t : ParallelTransition, gridPane : GridPane, a : int, b : int, c : int) : void
+ sortElementInBucket(t : ParallelTransition, gridPane : GridPane, j : int, i : int, count : int, timeJump : double) : void
+ mergeBucketsBackToArray(t : ParallelTransition, element : Node, gridPane : GridPane, j : int, i : int, count : int, timeJump : double) : void
+ bucketSort(t : ParallelTransition, buckets : Vector<Integer>[], gridPane : GridPane, timeJump : double) : void
+ initializeBuckets(t : ParallelTransition, buckets : Vector<Integer>[], gridPane : GridPane, timeJump : double) : void
+ sortBuckets(t : ParallelTransition, buckets : Vector<Integer>[], gridPane : GridPane, timeJump : double) : void
+ mergeBuckets(t : ParallelTransition, buckets : Vector<Integer>[], gridPane : GridPane, timeJump : double) : void
+ randomize(hbox : HBox, gridPane : GridPane) : void
+ onFinished(gridPane : GridPane) : void
MergeSort
- tmp : ColNode[]
+ Merge(list : ArrayList<ColNode>, p : int, q : int, r : int) : ArrayList<Transition>
+ mergeSort(list : ArrayList<ColNode>, p : int, r : int) : ArrayList<Transition>
AbstractSort
- COL_COUNT : int
- WINDOW_W : int
- WINDOW_H : int
- SELECTION : int
- COL_W : double
- NODE_COLOR : Color
- SELECT_COLOR : Color
- SORTED_COLOR : Color
- PIVOT_COLOR : Color
+ swap(list : ArrayList<ColNode>, a : int, b : int) : ParallelTransition
+ colorNodes(list : ArrayList<ColNode>, color : Color) : ParallelTransition
+ run(list : ArrayList<ColNode>) : ArrayList<Transition>
+ colorNodes(list : ArrayList<ColNode>, color : Color, a : int[]) : ParallelTransition
QuickSort
- transition : ArrayList<Transition>
+ quickSort(list : ArrayList<ColNode>, lo : int, hi : int) : void
+ partition(list : ArrayList<ColNode>, lo : int, hi : int) : int
BubbleSort
+ compare(list : ArrayList<ColNode>, a : int, b : int) : ArrayList<Transition>
components
array
BucketArray
- counter : int[]
- MAX : int
+ addNode() : void
+ getRandomNumber() : int
+ random() : void
CirArray
- cirArray : ArrayList<CirNode>
- lines : ArrayList<Line>
+ addCir() : void
+ displayHeap(pane : Pane) : void
+ getRandomNumber() : int
InitializeArr
+ initArr(n : int) : ArrayList<ColNode>
NodesList
- nodesList : ArrayList<Node>
- input : int[]
+ addNode() : void
+ displayNodesList(hbox : Hbox) : void
+ removeNumber(hbox : Hbox, index : int) : void
NumbArray
- arrayGUI : GridPane
- containers : VBox
- subSubContainers : Hbox
- subContainers : StackPane
- ids : Integer[]
- elements : ArrayList<StackPane>
+ displayNumbArray(cir : CirArray) : void
RadixArr
- inputPhase1 : int[]
- inputPhase2 : int[]
- inputPhase3 : int[]
- counter : Integer[]
- MAX : int
+ addNode() : void
+ getRandomNumber() : int
+ random() : void
node
ColNode
+ moveX(dist : double) : TranslateTransition
CirNode
- id : int
- radius : double
- index : int
+ CirNode(id : int, color_1 : Color, color_2 : Color) : void
+ getCenterOfCirNode_X() : double
+ getCenterOfCirNode_Y() : double
Node
- rec : Rectangle
- text : Text
+ Node(id : int) : void
+ Node() : void
+ Node(id : int, color : Color) : void
+ Node(color : Color) : void
MainMenu
+ start(primaryStage : Stage) : void
+ main(args : String[]) : void
MainMenuController
- scene2 : Scene
- Submitbtn : Button
- sorts : ChoiceBox<String>
+ SubmitBtnHandler(evt : ActionEvent) : void
+ initScene(i : int) : void
scenes
bubble_quick_merge
SortSceneController
- mainScene : Scene
- isStop : boolean
- sq : SequentialTransition
- corgitt : TranslateTransition
- finishbtn : Button
- pausebtn : Button
- playbtn : Button
- randombtn : Button
- backbtn : Button
- display : Hbox
- corgi : ImageView
+ initialize(arg0 : URL, arg1 : ResourceBundle) : void
+ playBtnHandler(evt : ActionEvent) : void
+ randomBtnHandler(evt : ActionEvent) : void
+ backBtnHandler(evt : ActionEvent) : void
+ pauseBtnHandler(evt : ActionEvent) : void
+ finishBtnHandler(evt : ActionEvent) : void
bucket
BucketSortGUI
- button_1 : Button
- button_2 : Button
- button_3 : Button
- button_4 : Button
- button_5 : Button
- bucketArr : BucketArr()
- bucketList : int[]
- buckets : Vector<Integer>[]
- isStop : boolean
- buttons : List<Button>
- hbox : Hbox
- gridPane : GridPane
+ createScene() : void
radix
RadixSortGUI
- button_1 : Button
- button_2 : Button
- button_3 : Button
- button_4 : Button
- button_5 : Button
- isStop : boolean
- digitsList : int[]
- digits : Vector<Integer>[]
- hbx : Hbox
- gridPane : GridPane
- buttons : List<Button>
+ getMax(arr : int[]) : int
+ createScene() : void
heap
HeapSortGUI
- button_1 : Button
- button_2 : Button
- button_3 : Button
- button_4 : Button
- button_5 : Button
- buttons : List<Button>
- isStop : boolean
- cirGroup : CirArray
+ createScene() : void

(To see the class diagram more clearly, please open the .png file we have attached, or open the source file in astah.)

Analysis of the classes:

1. Package “components”	
Sub-package “node”	
Class	Role
Node	This class extends the class StackPane, to initialize the squares on screen
CirNode	This class extends the class StackPane, to make a node consisting of one circle and the text
ColNode	This class extends the class rectangle, used to show columns
Sub-package “array”	
Class	Role
BucketArr	inherit from the nodelist, override the addNode, because the random of the bucket sort is different (visualization efficiency)
CirArray	To initialize an array list of <CirNode> objects array list of <Line> objects
InitializeArr	To initialize an array list of <ColNode> objects
NodesList	To initialize an array list of the Nodes, containing the array of values we need to sort
NumbArray	To display the the array of numbers down below in the screen

RadixArr	This class extends the class NodeList, override the addNode, because the random of the radix sort is different (visualization efficiency)

2. Package “sorts”	
Class	Role
AbstractSort	An abstract class that will be inherited by BubbleSort, MergeSort, and QuickSort, containing all the common attributes and methods for these three classes.
BubbleSort	Containing the bubble sort algorithm, extends AbstractSort
BucketSort	Containing the bucket sort algorithm
HeapSort	Containing the heap sort algorithm
MergeSort	Containing the merge sort algorithm, extends AbstractSort
QuickSort	Containing the quick sort algorithm, extends AbstractSort
RadixSort	Containing the radix sort algorithm

3. Package “scenes”	
Sub-package “bubble_quick_merge”	
SortSceneController	Initializes the columns on the screen for all three following sorting algorithms: Bubble, Quick, and Merge

	Sort, as well as handles all the events. The GUI design is in the fxml file which is from the same sub-package
Sub-package "bucket"	
BucketSortGUI	Initializes the bucket list and the nodes on the screen, handles all the events, as well as contains the GUI design
Sub-package "heap"	
HeapSortGUI	Initializes the circle nodes and the number array on the screen, handles all the events, as well as contains the GUI design
Sub-package "radix"	
RadixSortGUI	Initializes the digit list and nodes on the screen, handles all the events, as well as contains the GUI design

II. Run the program

How to run the executable file from command lines:

<u>Step 1</u>: Make sure you already have the **javafx library** installed in your system. The program can not run without it.

<u>Step 2</u>: After extracting the zip file, access the .jar file in its folder, The relative path from the archive to the **.jar** file is "OOPproject-Group2-Topic5\src\out\artifacts\SortingAlgorithms"

Right-click and choose the option "**Open with cmd**".

<u>Step 3</u>: The command format is as follow:

```
java [VM Arguments] -jar OOPproject-Group2-Topic5.jar
```

[VM Arguments] format:

```
 --module-path "javafx lib file path" --add-modules=javafx.controls,javafx.fxml
```

For example, my javafx lib file path is “D:\DOCUMENT\2020-2021 year\20202 sem\OOP Java\openjfx-16_windows-x64_bin-sdk\javafx-sdk-16\lib”

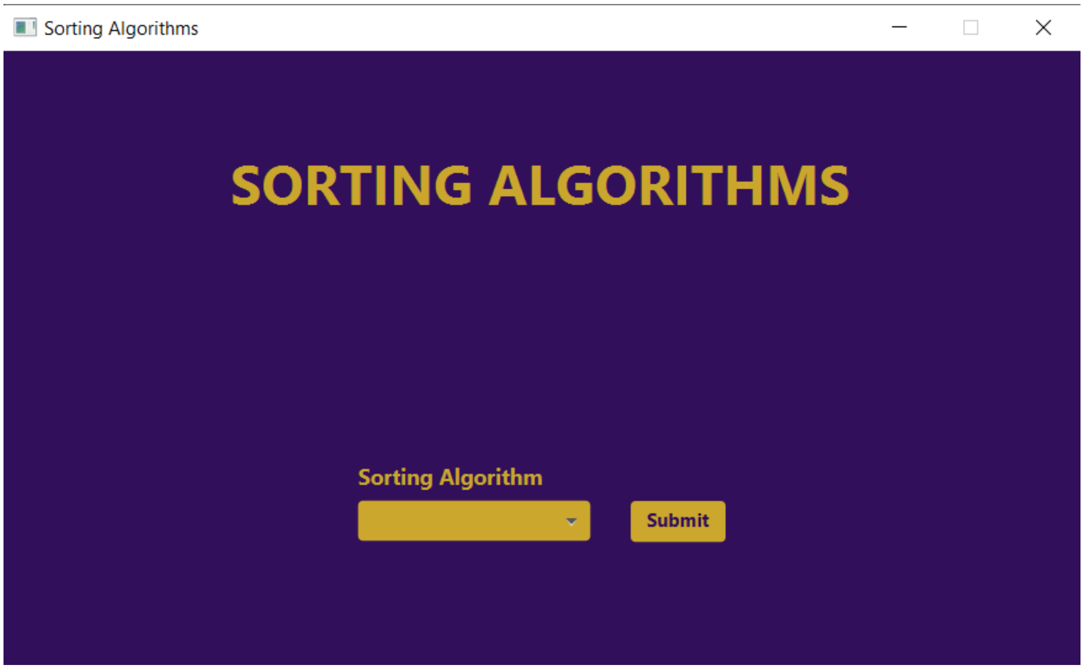
My [VM Arguments] would be:

```
 --module-path "D:\DOCUMENT\2020-2021 year\20202 sem\OOP Java\openjfx-16_windows-x64_bin-sdk\javafx-sdk-16\lib" --add-modules=javafx.controls,javafx.fxml
```

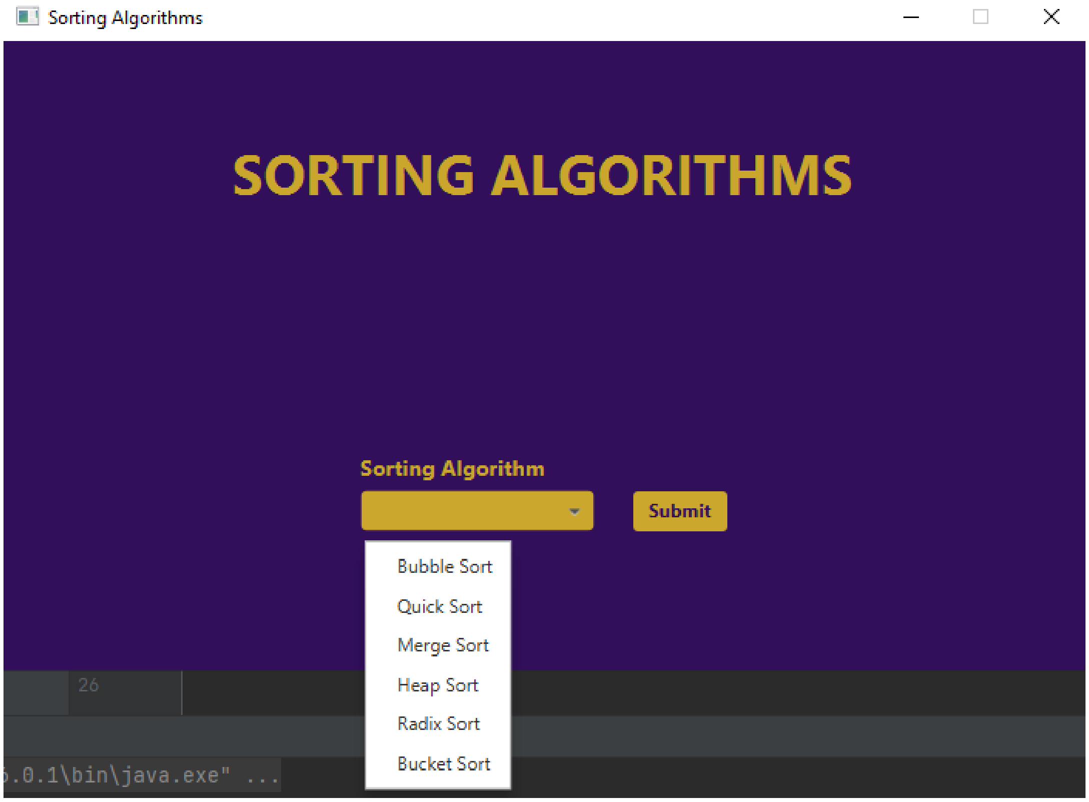
My full command line would be:

```
java  --module-path "D:\DOCUMENT\2020-2021 year\20202 sem\OOP Java\openjfx-16_windows-x64_bin-sdk\javafx-sdk-16\lib" --add-modules=javafx.controls,javafx.fxml -jar OOPproject-Group2-Topic5.jar
```

When the program starts running, a main **menu window** will pop up

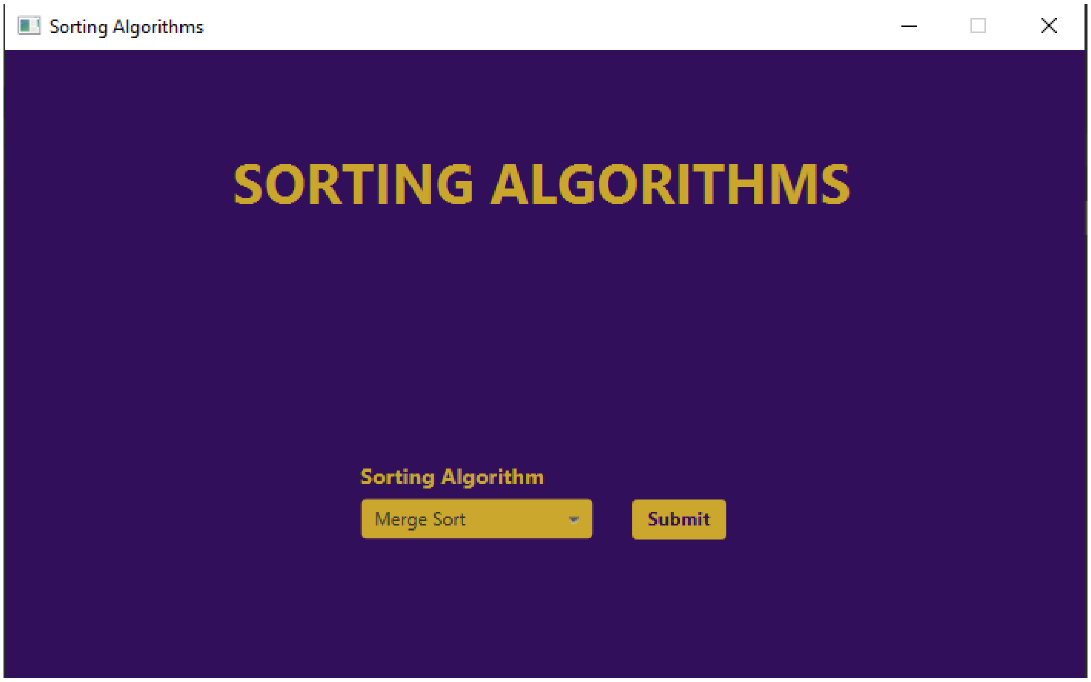


The user can **choose the sorting method** they want to visualize



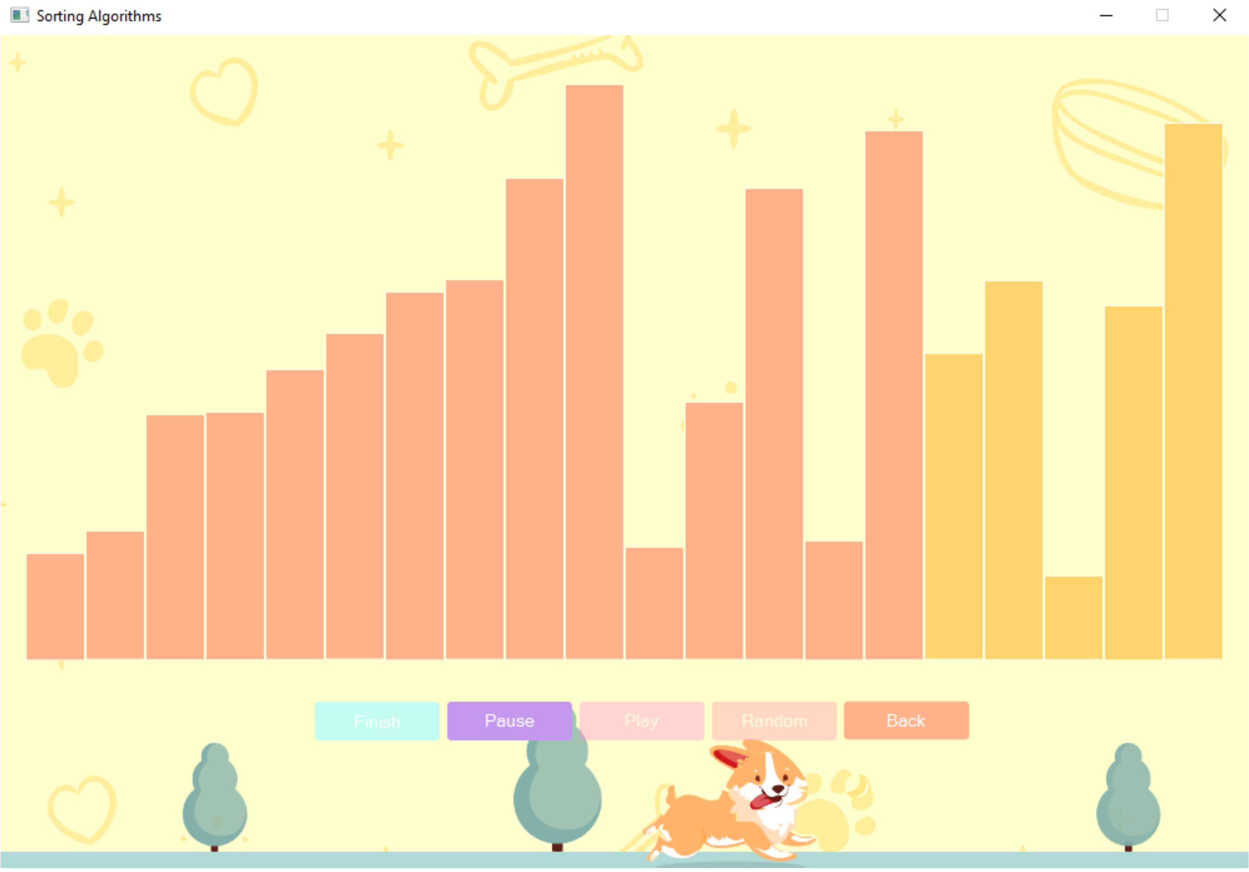
```
.0.1\bin\java.exe" ...
```

After choosing, for example, “merge sort”, **click Submit**



Now the sorting window will pop up, in this window there are 5 different functional buttons, “**Start**”, “**Random**”, “**Pause**”, “**Finish**”, and “**Back to menu**”

1. **Start**: *start* sorting
2. **Pause**: *pause* sorting, and to *continue* sorting after pausing
3. **Finish**: show the sorting *result* immediately
4. **Random**: generate a new *random* set of values to sort again
5. **Back to menu**: go *back* to main menu window



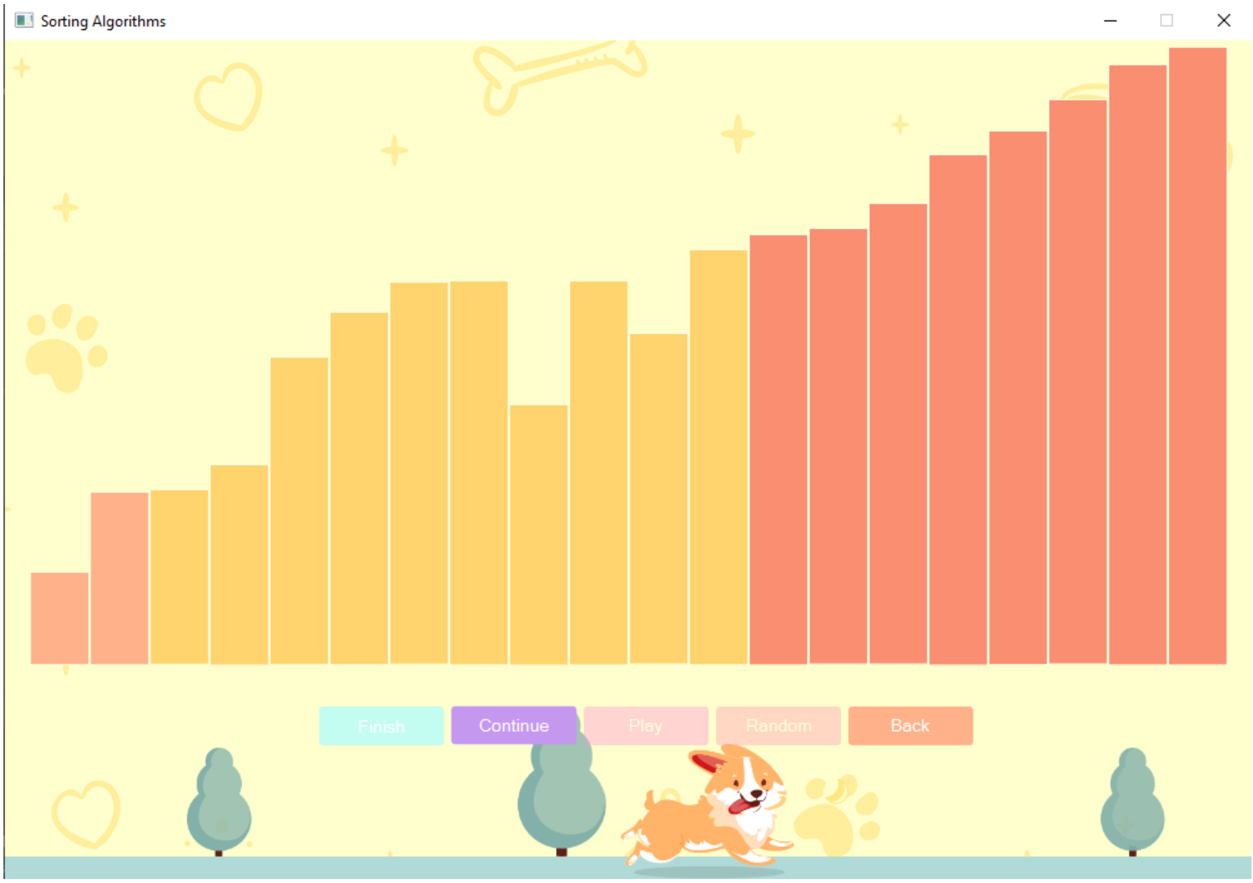
Repeat the steps above to try other sorting methods

III. Analysis on each sorting algorithm

1. Bubble Sort

This is one of the most simple sorting algorithms out there. All we need to do is track two **adjacent** values, or in this case, columns, to see if the first column is shorter than the one behind it. If that is true, then move on to the next pair, if not, then those two columns **swap** positions.

After the first loop, the highest column will be moved to the bottom of the array, after the second loop, the second highest column will be moved to the second position from the bottom, and so on,... All these sorted columns at the back will turn red.

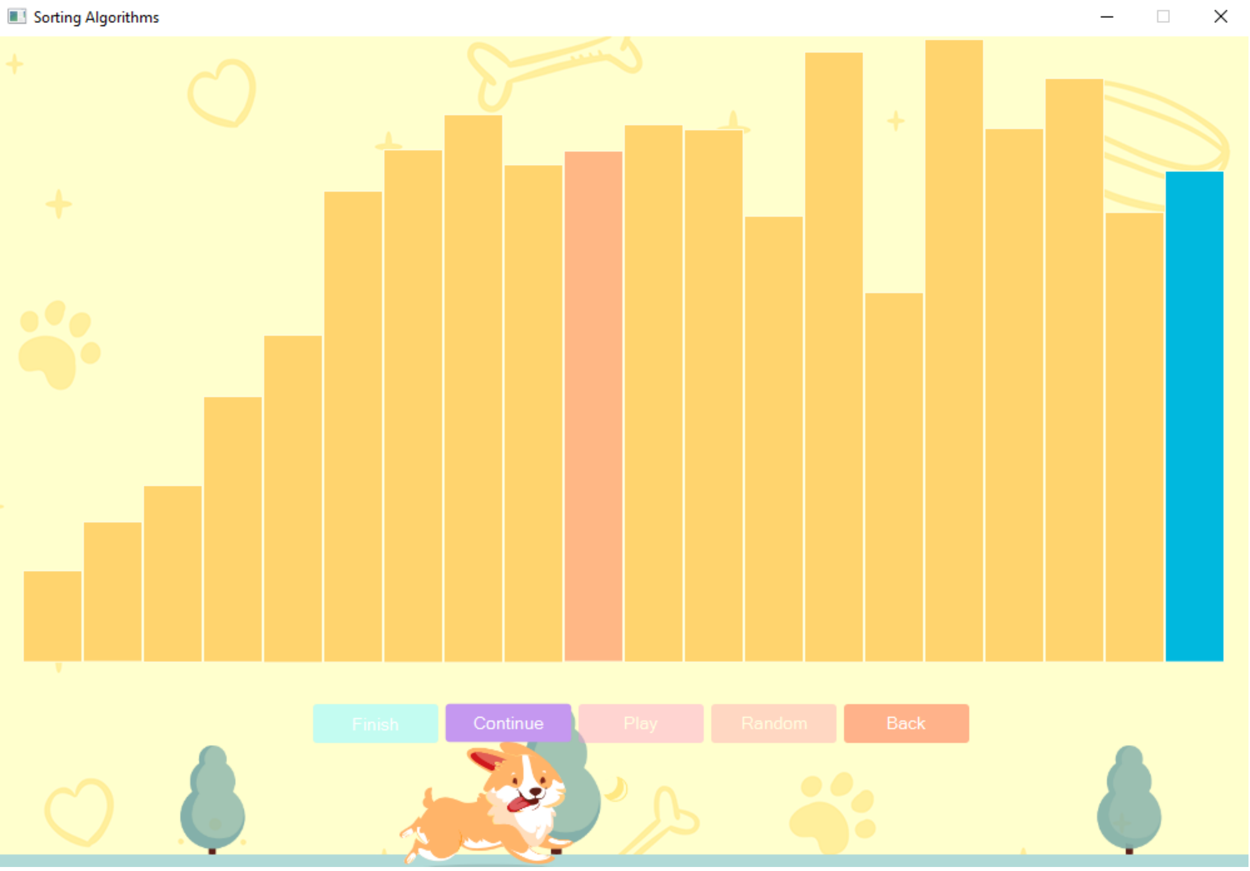


2. Quick Sort

Two key words we need to keep in mind while implementing Quick Sort algorithm are “**pivot**” and “**partition**”. In this program, the **last** element is automatically picked as the pivot, marked with the blue color.

This algorithm is of Divide and Conquer type. The main idea is that the pivot will **divide** the array into two halves, with the left half being smaller than the pivot, while the right half being bigger.

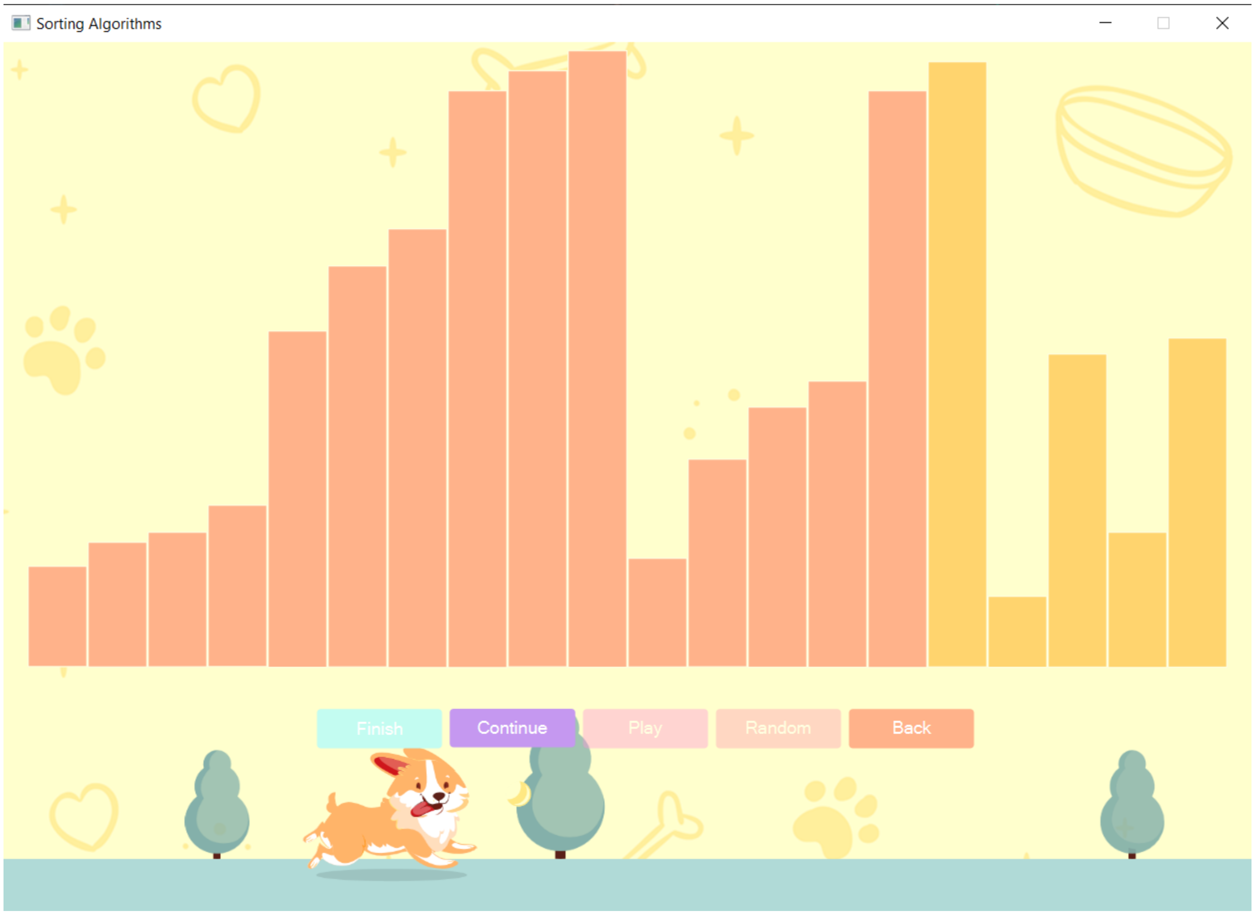
The “**partition**” method will return the index of where the chosen “**pivot**” should be in the array, by swapping the column at index i and j (j is the looping iteration index, i starts from the first index and increments by 1 whenever the column at j is smaller than the pivot). Then keep implementing quick sort for the left half and the right half.



3. Merge Sort

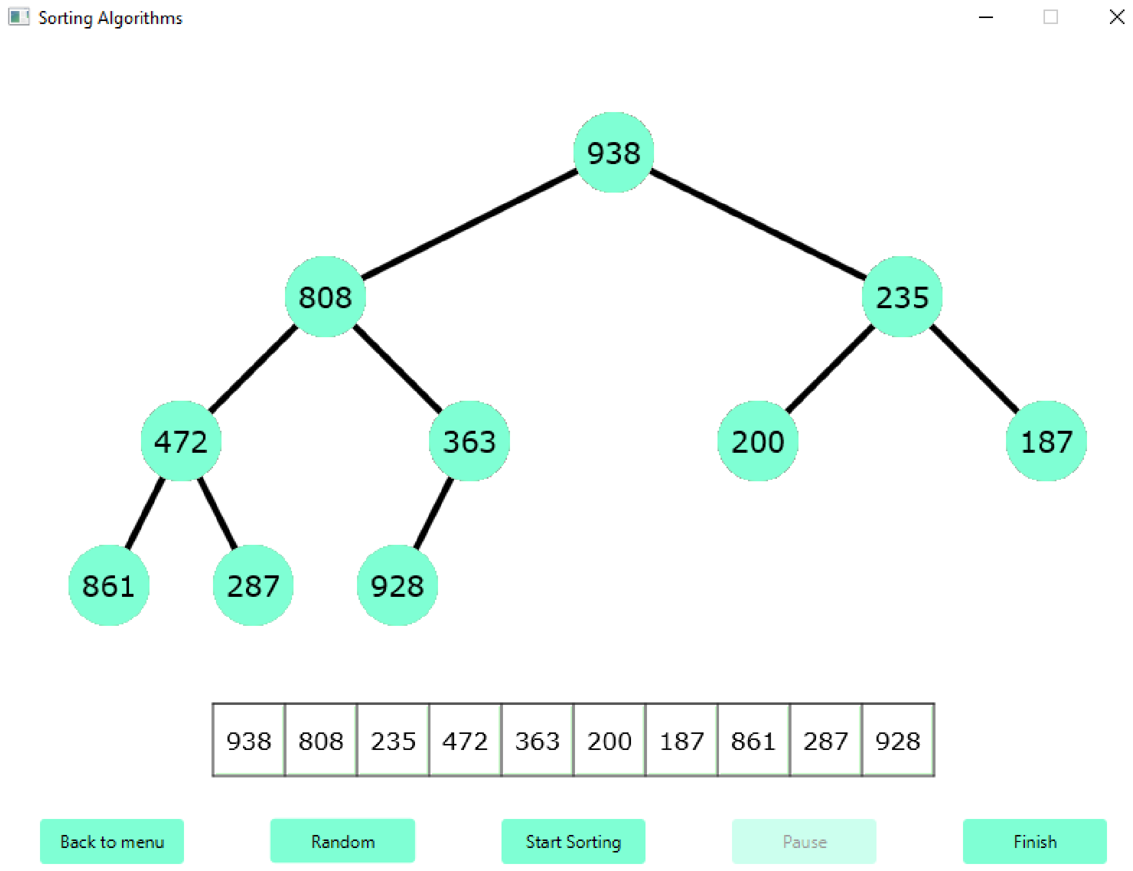
Similar to Quick Sort, Merge Sort is also a Divide and Conquer algorithm, it **divides** the array into two halves, then repeatedly calls the merge sort function upon each half until they are both sorted, then both halves would be **merged** into each other.

The mergeSort function is called recursively until the size of the "half" gets to 1, then the merging starts. When merging two arrays of sorted columns, we start both indexes from the first element, compare those two elements, then increment the iterator index of the array that contains the smaller element in the previous comparison.



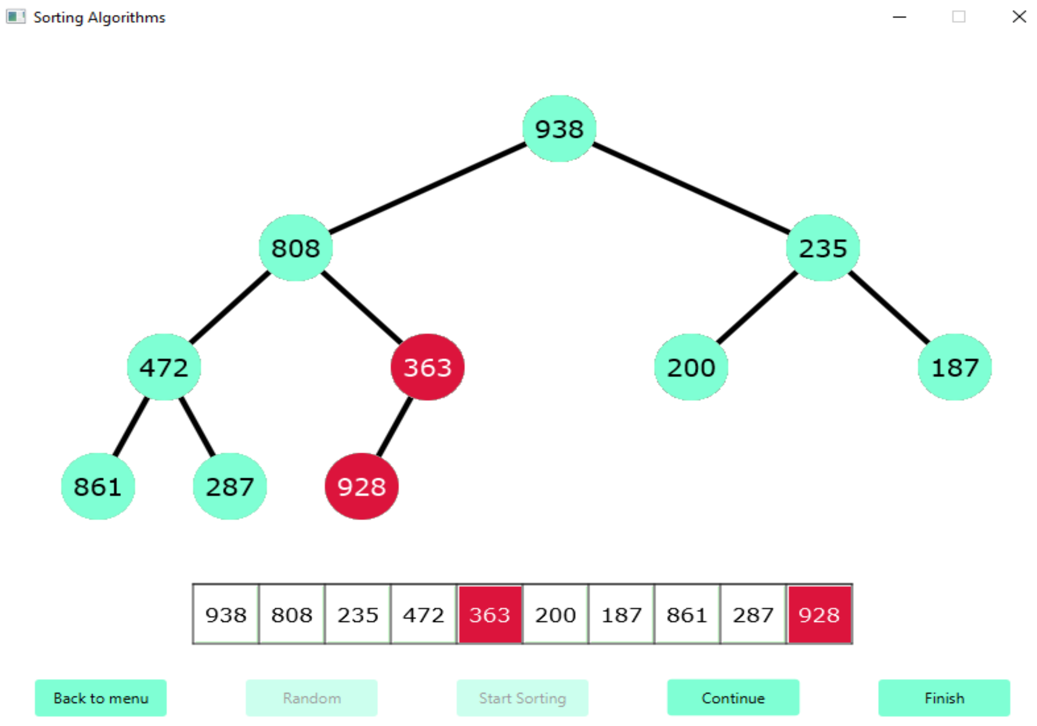
4. Heap Sort

First, the array of random numbers will be put into a **heap** - a tree-based data structure in which the tree is a complete **binary** tree.



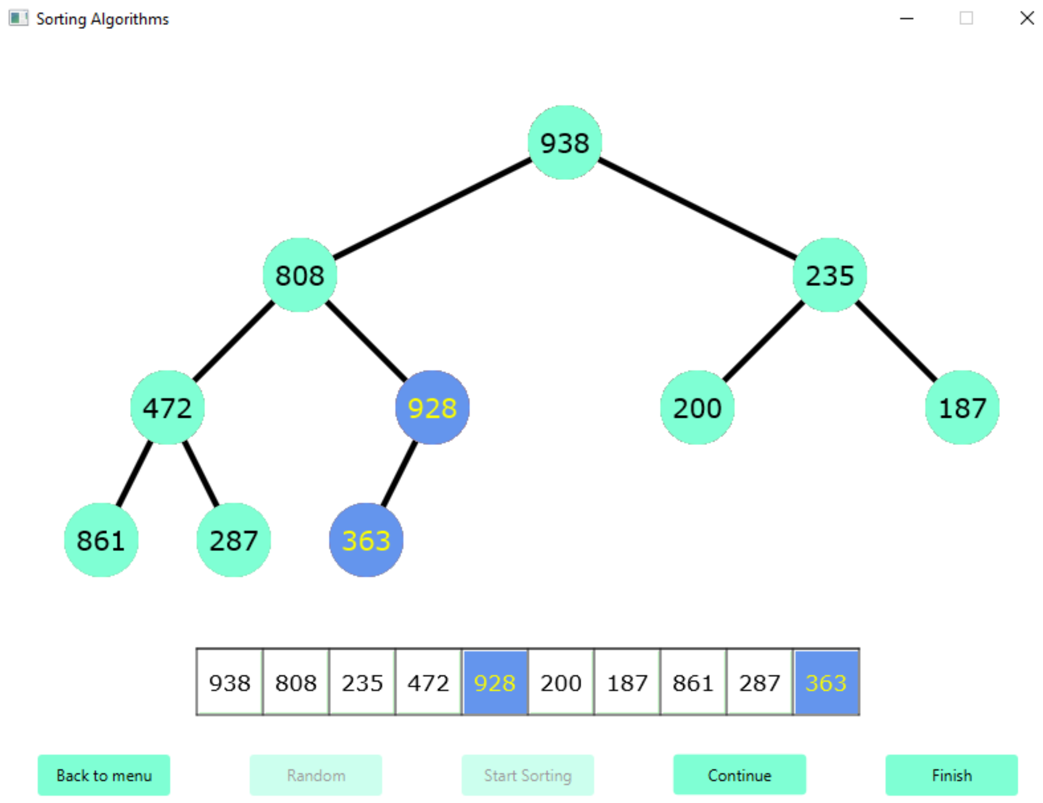
938	808	235	472	363	200	187	861	287	928

The HeapSort will start from the **last** parent node (downward and rightmost from the top of the tree), when checking a node, the node will turn red. Here in this case, the last parent node is 363, and its child node is 928.



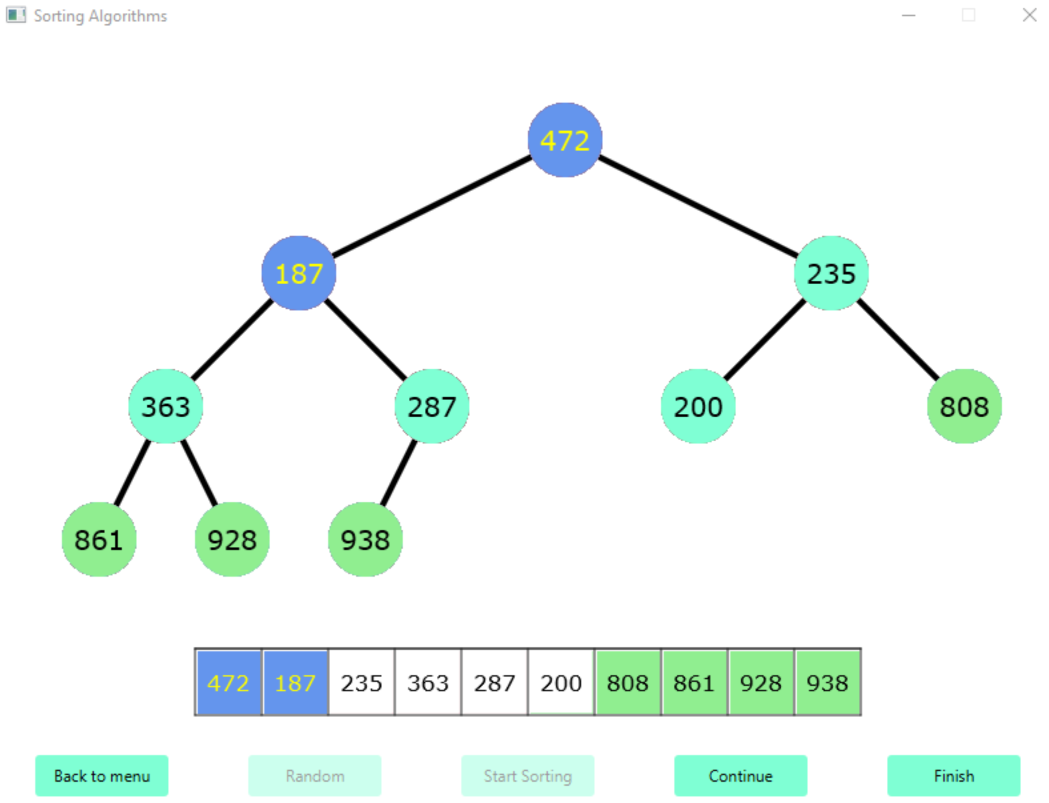
938	808	235	472	363	200	187	861	287	928

If that parent node happens to be smaller than the bigger child node out of the two child nodes, we will **swap** the parent node with that child node. (Or in this case, swap with its only child node). Now they will turn blue.



938	808	235	472	928	200	187	861	287	363

After each loop the goal of the program is to build a **max heap**, which is a binary tree in which the parent is always larger than the child nodes. One loop ends when the positions of the root node and the last node are swapped. The sorted nodes turn green, and BuildMaxHeap continues in the following loops, until the whole array is sorted.

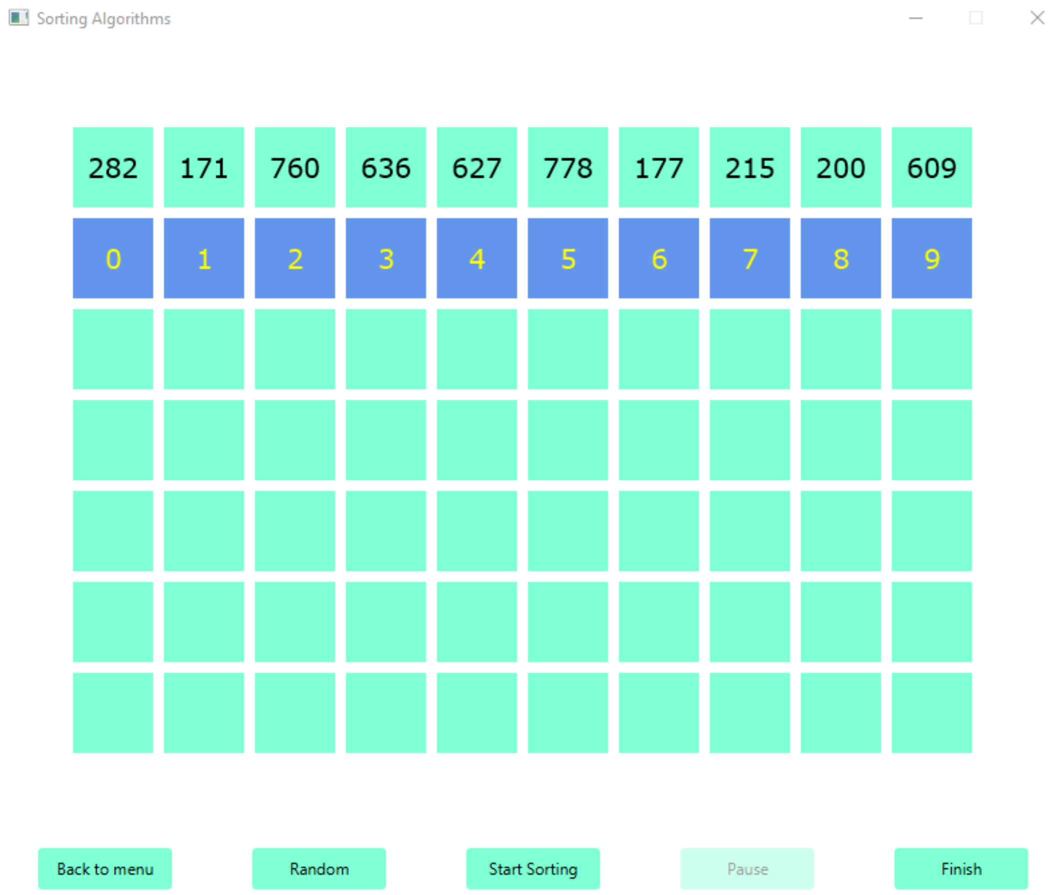


472	187	235	363	287	200	808	861	928	938

5. Radix Sort

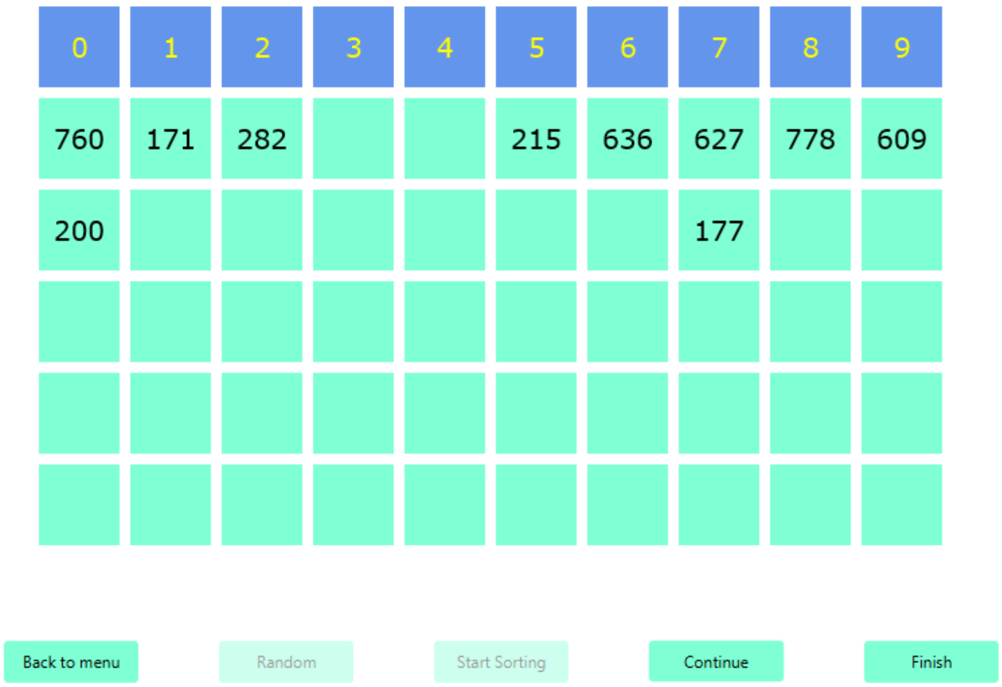
For this sort algorithm we will only consider the value from 0-999 to have a better visualization of the algorithm.

We have an array of unsorted values.



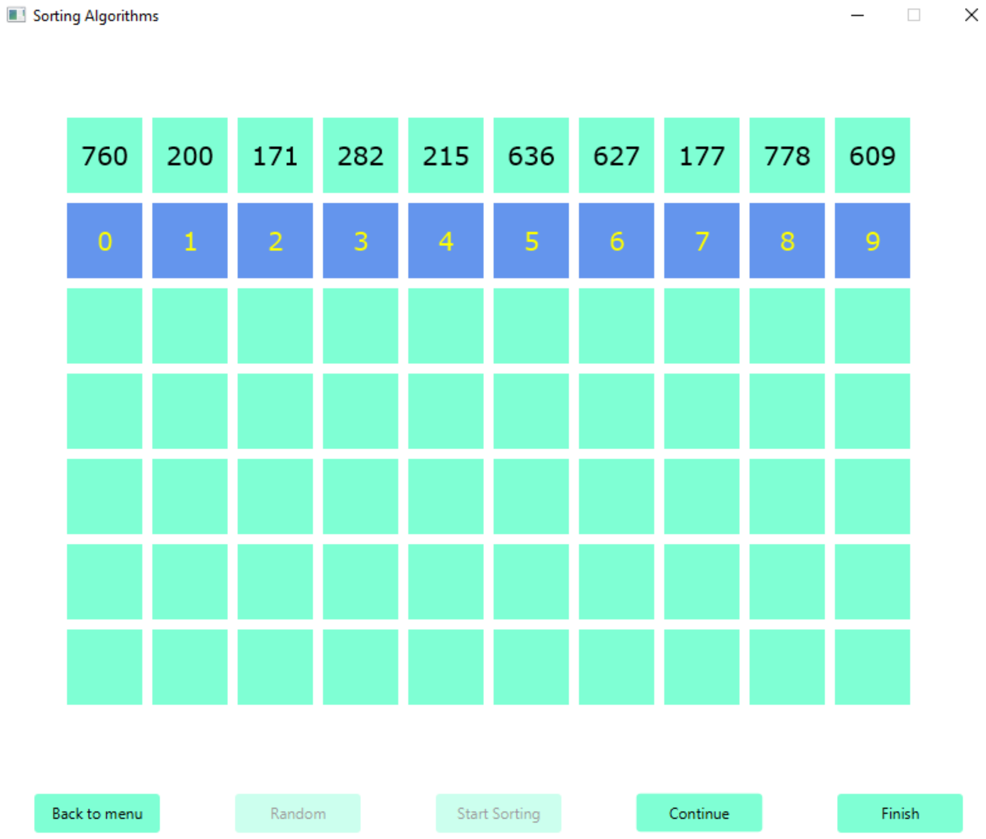
282	171	760	636	627	778	177	215	200	609
0	1	2	3	4	5	6	7	8	9

First, we traverse through the array in its order, and compare their **one’s digits**, and put them in the respective columns under the **radix** value. For example, the numbers that have the last digit ‘6’ will be put under the 6 column.



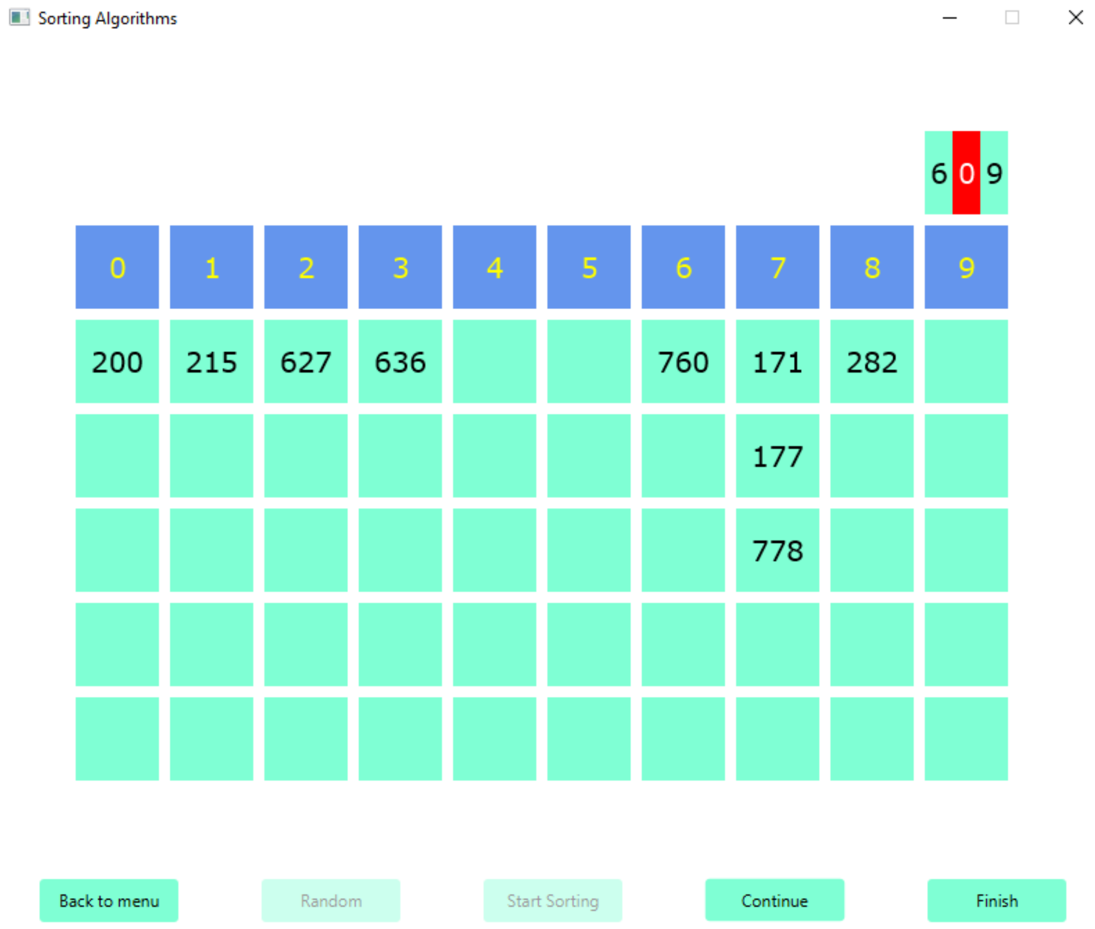
0	1	2	3	4	5	6	7	8	9
760	171	282			215	636	627	778	609
200							177		

Then the numbers will be sequentially put back into a new array



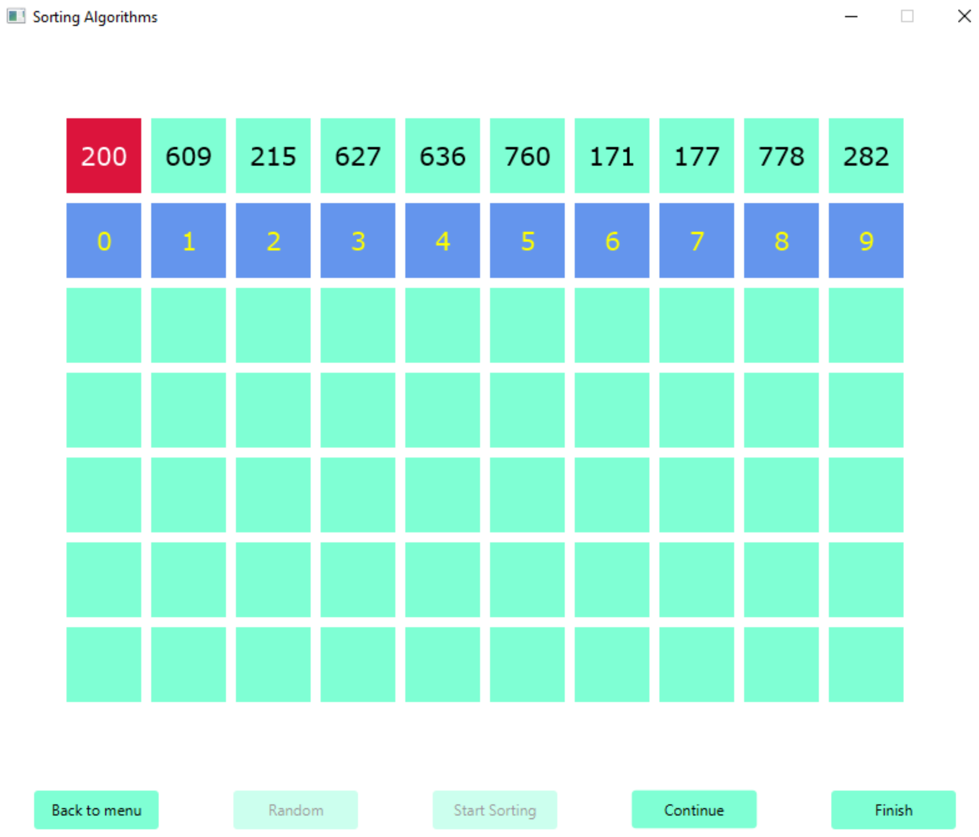
760	200	171	282	215	636	627	177	778	609
0	1	2	3	4	5	6	7	8	9

Repeat the first step, but with the **ten's digit** (note: whenever a digit is checked, it turns red)



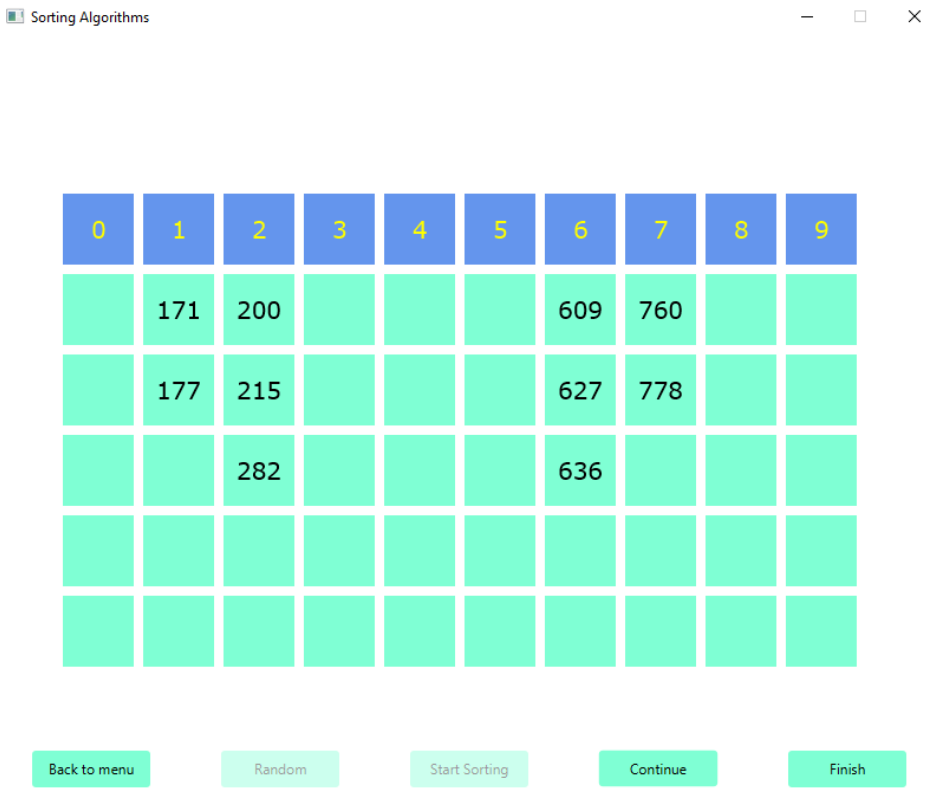
609

0	1	2	3	4	5	6	7	8	9
200	215	627	636			760	171	282	
							177		
							778		



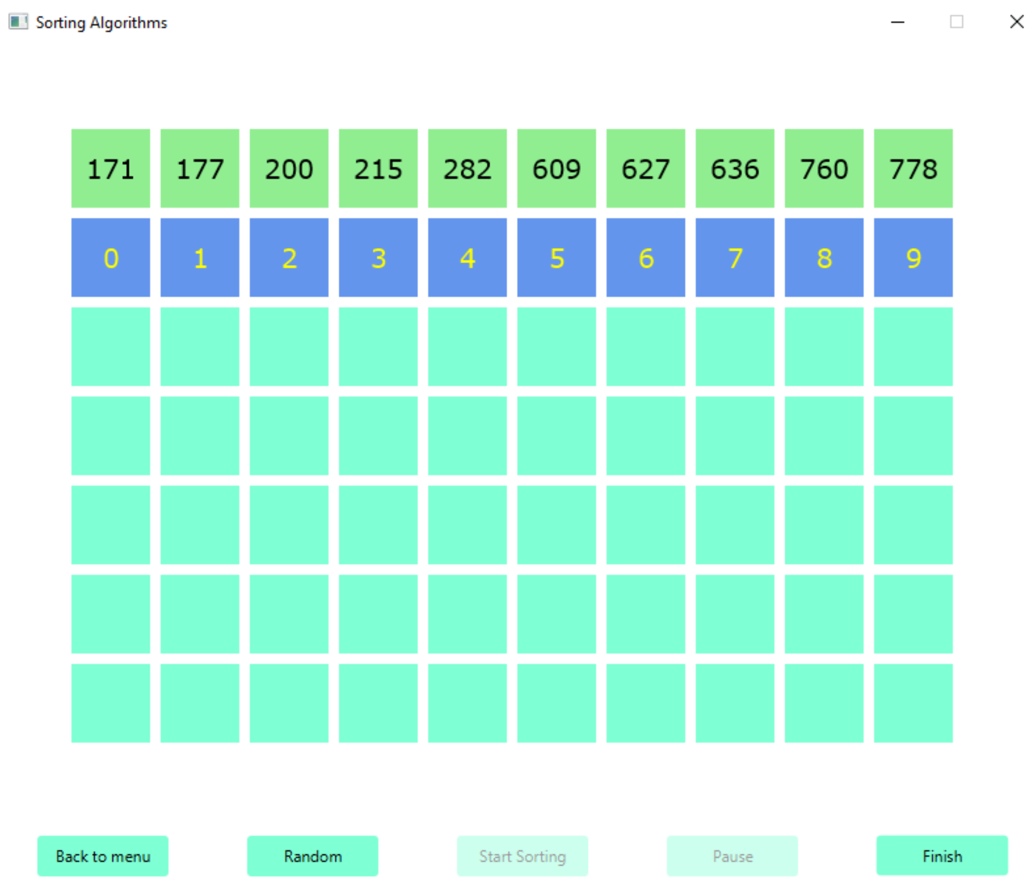
200	609	215	627	636	760	171	177	778	282
0	1	2	3	4	5	6	7	8	9

And again with the **hundredth's digit**



0	1	2	3	4	5	6	7	8	9
	171	200				609	760		
	177	215				627	778		
		282				636			

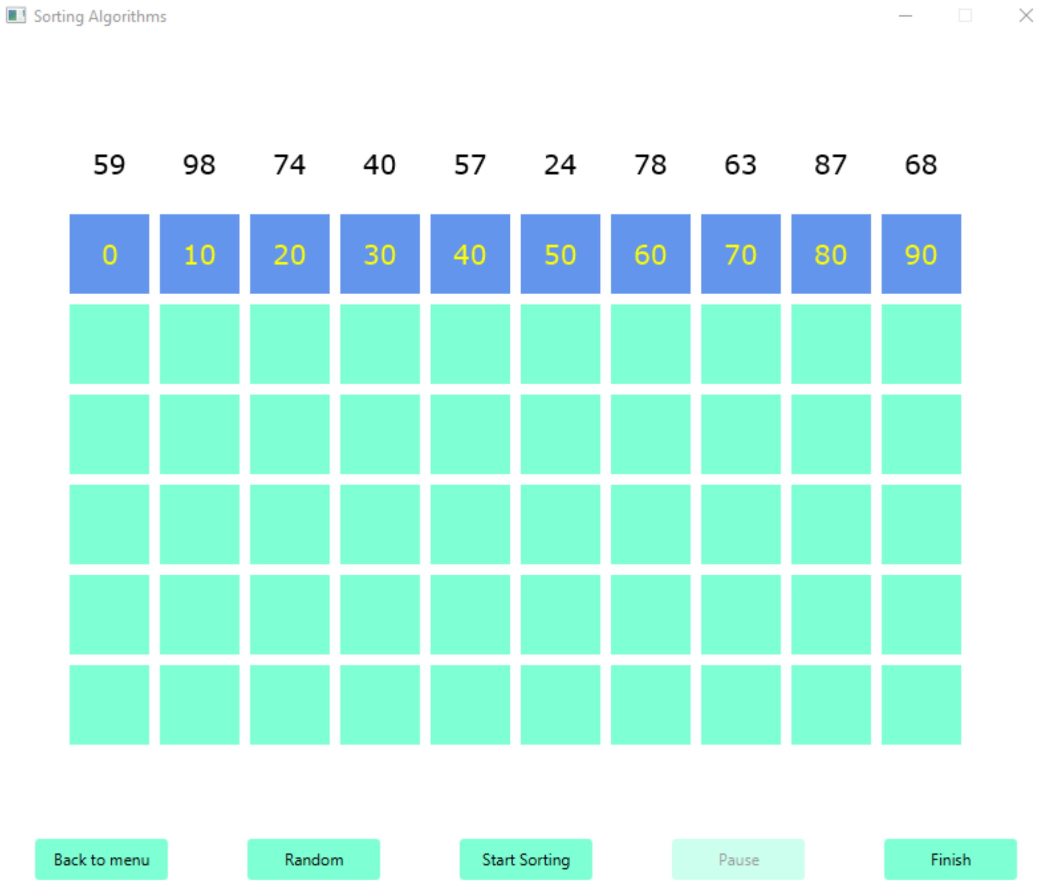
And we attain the sorted array!



171	177	200	215	282	609	627	636	760	778
0	1	2	3	4	5	6	7	8	9

6. Bucket Sort

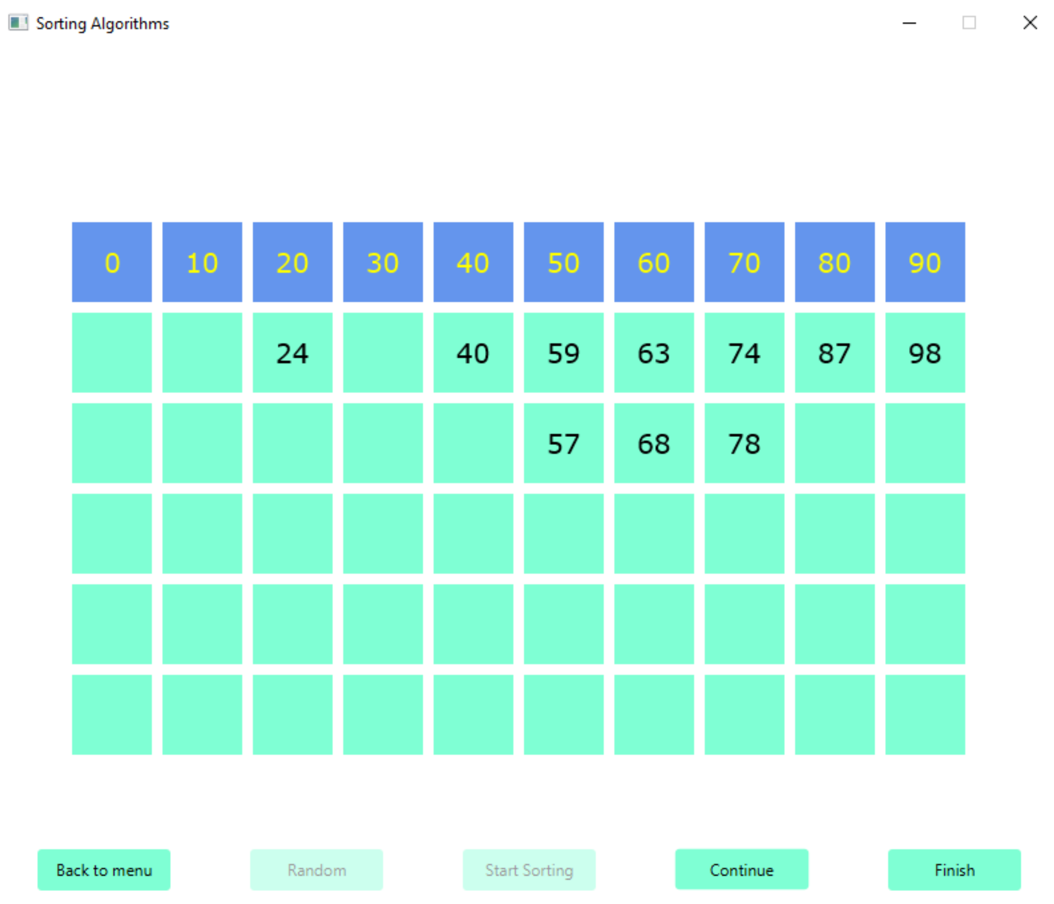
In this sorting algorithm, we create **buckets** to put elements into.



59 98 74 40 57 24 78 63 87 68

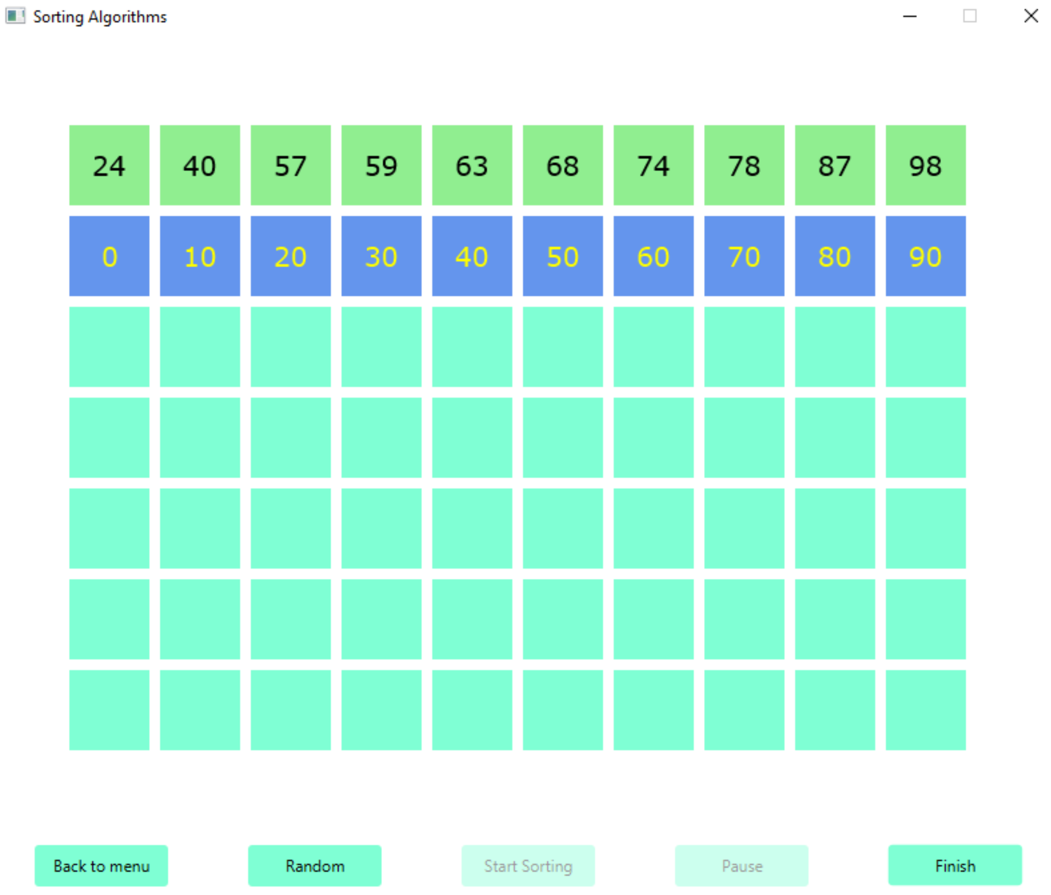
0	10	20	30	40	50	60	70	80	90

The rule is simple, bucket 0 will contain values from 0 to 9, bucket 10 will contain values from 10 to 19, bucket 20 will contain values from 20 to 29, and so on. The values in this program will be limited from 0 to 99 to achieve visualization efficiency.



0	10	20	30	40	50	60	70	80	90
		24		40	59	63	74	87	98
					57	68	78		

After that, the values in each bucket will be **sorted.** After sorting, they will be **merged** back into a fully sorted array.



Sorting Algorithms

24	40	57	59	63	68	74	78	87	98
0	10	20	30	40	50	60	70	80	90

END OF REPORT