Sorting algorithm visualization

H09

IWASAKI, Keita 20684826 kiwasakiaa@connect.ust.hk

LEE, Young Kyu 20219190 ykleeac@connect.ust.hk

Table of Contents

Task Assignment 2
Testing Environment 2
Project objectives 2
Features incorporated 3
OOP design 6
Implementation 7
Data structures used 20
Conclusion 21

Task Assignment

- IWASAKI, Keita : UI and controller (MainWindow class / Paint class)
- Lee, Young Kyu : UI and sorting logic (Sorting class)

Testing Environment

- MacOS Catalina
- Qt Creator 4.10.2
- Based on Qt 5.13.2 (Clang 10.0 (Apple), 64 bit)

Project objectives

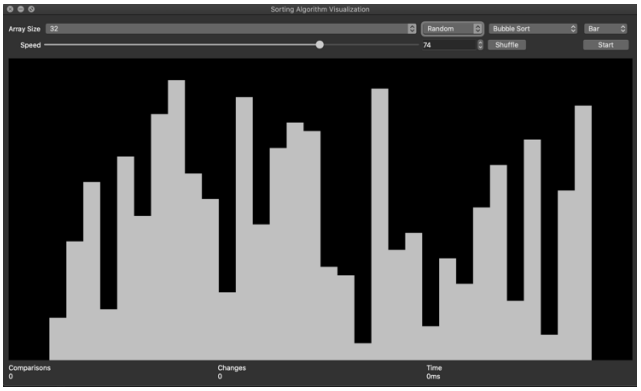
Since there were many types of sorting algorithms, it is highly important for us to know the difference in simplicity and efficiency. We choose Qt Instead of using console because users can intuitively manipulate the type of algorithm. This Qt application visualizes the various type of sorting algorithms with different speed and different types of visualization using with sounds. Users can choose the sorting algorithm, number of elements, speed and types of visualization.

Features incorporated

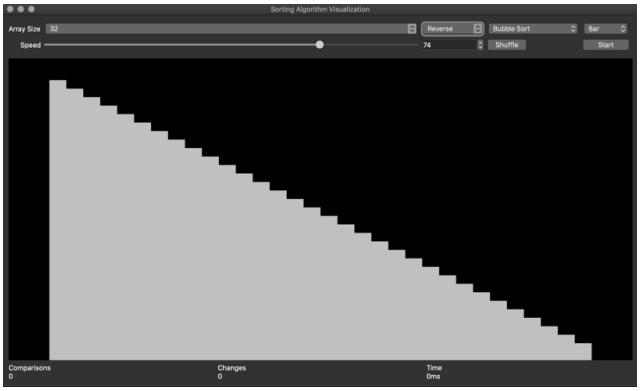
- **Various shuffle types**

  Users can select how to shuffle elements before the sorting.

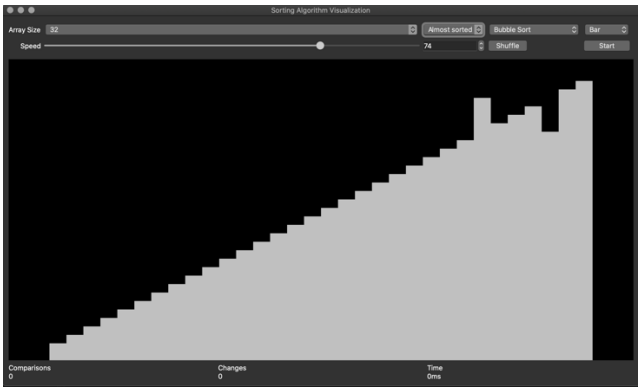
  - Random sort



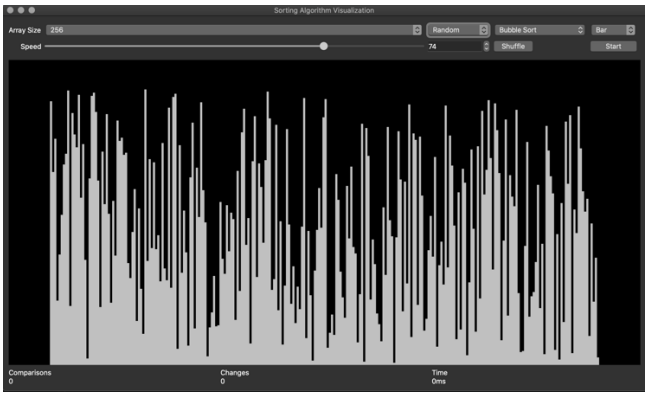
  - Reverse sort



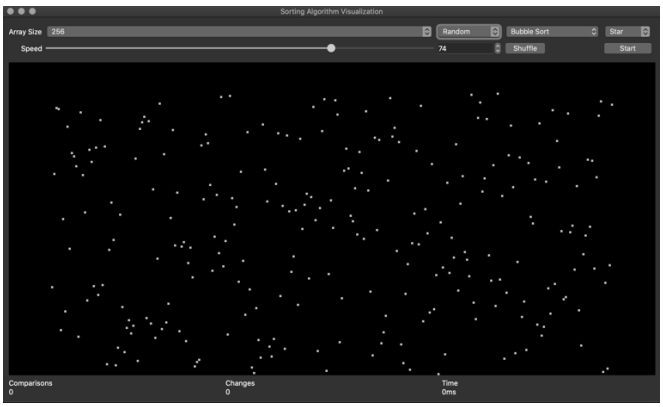
  - Almost sorted



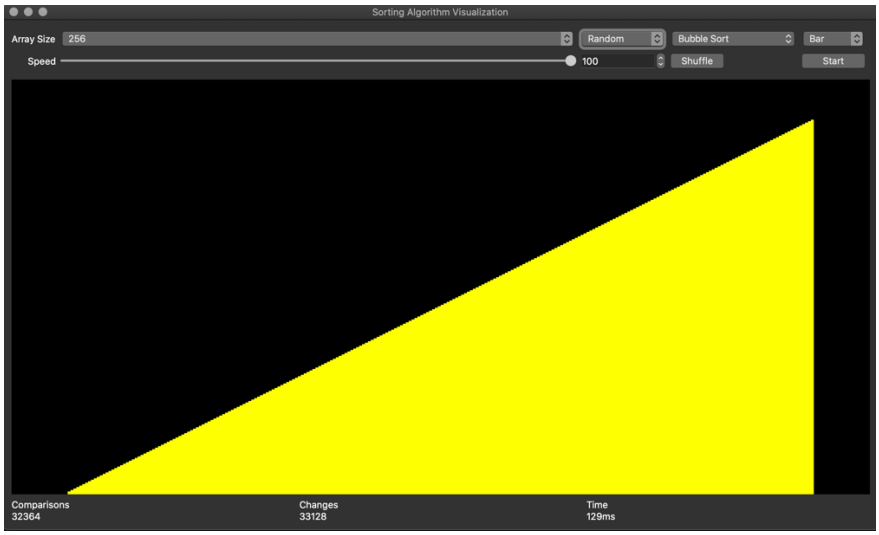
- **Shape of the elements**
    - Bar



    - Star

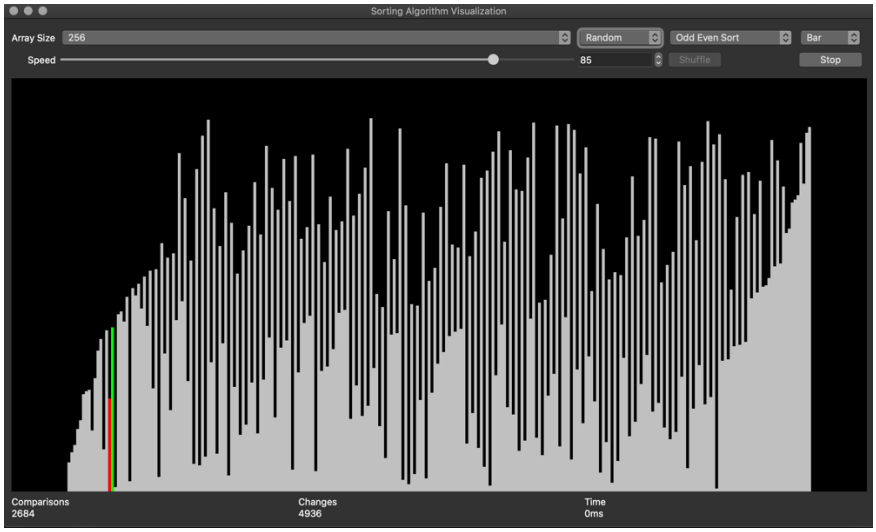


- **Information about each sorting**
    1. Number of comparisons
    2. Number of changes
    3. Execution time



- Color bars

  Moving bars are colored differently



- **Number of the elements**

  Users can select from the following options

  2, 4, 6, 8, 16, 32, 64, 256, 512, 1024

- **Speed of the sorting**

  Users can select an integer between 0~100

- **Sorting algorithm**

  Users can select a sorting algorithm from the following

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Merge Sort
5. Heap Sort
6. Quick Sort
7. Radix Sort (Base 2)
8. Radix Sort(Base 10)
9. Counting Sort
10. Gnome Sort
11. Cocktail Sort
12. Comb Sort
13. Odd Even Sort

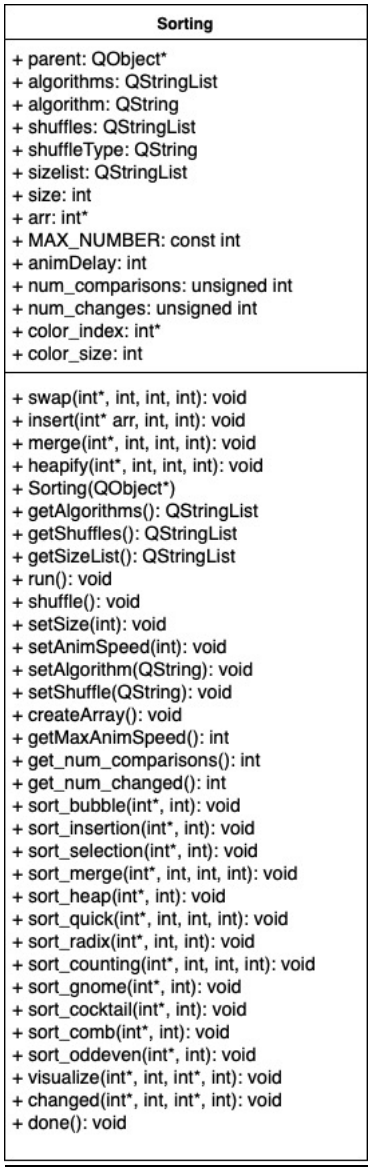
OOP design

There are three classes defined in our projects: MainWindow, Paint and Sorting.

Paint class's main work is drawing the elements. Sorting class is responsible for the sorting logic, all the sorting algorithms are written in this class. MainWindow is the class to work as a mediator between UI and other classes. It receives the signal from UI components and notifies to other classes accordingly and vice versa.

Implementation

Sorting



Sorting class is the class where the actual sorting procedures take place. It creates, shuffles, and sorts integer array of specific size. It is a member variable of MainWindow class.

Data members explanation

- `QStringList algorithms:` List of available sorting algorithms in this project.
- `QString algorithm:` The algorithm which is chosen currently.

- `QStringList shuffles:` List of available options for the shuffling of the array.
- `QString shuffleType:` The shuffling option which is chosen currently.
- `QStringList sizelist:` List of available size of the array.
- `int size:` Size of `arr`.
- `int* arr:` The actual array which sorting and shuffling procedures take place.
- `const int MAX_NUMBER:` An upper bound to the elements in the array.
- `int animDelay:` Delay between each change in the visualization
- `unsigned int num_comparisons:` Number of comparisons made in the sorti ng procedure. Note that since Radix sort and Counting sorts are non-comparison base d sorting algorithm, they do not increase `num_comparisons`.
- `unsigned int num_changes:` Number of times where the elements of the arra y are changed.
- `int* color_index:` An array of indices to be colored in the visualization.
- `int color_size:` Size of `color_index`.

Member functions explanation

- `void swap(int* arr, int size, int i, int j):` Swap `arr[i]` an d `arr[j]` and increase `num_changes` by 2. Used in most in-place sorting algorith m.
- `void insert(int* arr, int size, int i):` Search for correct relative position for `arr[i]` and insert `arr[i]` into that position, and increase the `num_ch anges` and `num_comparisons` accordingly. In the searching of the position, linea r search is used.
- `void merge(int* arr, int size, int start, int end):` For suba rray of `arr` (from index `start` to `end`), split it into two smaller array of similar siz

e, and merge them with typical merging method.

- `void heapify(int* arr, int size,int heapsize, int i):` Heap ify a subtree rooted with node `i`, with size `heapsize`. Max heap is used in this func tion.
- `Sorting(QObject* parent):` A constructor for `Sorting` class.
- `QStringList getAlgorithms():` Returns `algorithms`, which is used in `c omboBoxAlgorithm` in `MainWindow` class.
- `QStringList getShuffles():` Returns `shuffles`, which is used in `combo BoxShuffle` in `MainWindow` class.
- `QStringList getSizeList():` Returns `sizelist`, which is used in `combo BoxArraySize` in `MainWindow` class.
- `void run():` Starts sorting `arr` with chosen algorithm.
- `void shuffle():` Shuffles `arr` with chosen shuffling option.
- `void setSize(int size):` Changes size of `arr`. For size 1024, there will be some repeated elements, where for other sizes, every element is unique.
- `void setAnimSpeed(int speed):` Changes speed of visualization.
- `void setAlgorithm(QString option):` Changes sorting algorithm to be a pplied on `arr`.
- `Void setShuffle(QString option):` Changes shuffling option. `"Rando m"` and `"Reverse"` are self explanatory, where `"Almost sorted"` is different f rom typical definition, it swap two random elements in sorted array. (If we use typica l definition, which every element is at most 1 unit away from it's correct position, it i s almost impossible to see the difference of almost sorted array and sorted array when the size of array is large, thus this alternative option is used.)
- `void createArray():` Creates an array to be shuffled and sorted.

- `int getDefaultAnimSpeed()`: Returns `DEFAULT_ANIM_SPEED`. It is used for `spinBoxAnimSpeed` and `SliderSpeed` in `MainWindow` class.
- `int getMaxAnimSpeed()`: Returns `MAX_ANIM_SPEED`. It is used for `spinBoxAnimSpeed` and `SliderSpeed` in `MainWindow` class.
- `unsigned int get_num_comparisons()`: Returns `num_comparisons`. It is used in `labelComparisons` in `MainWindow` class.
- `unsigned int get_num_changes()`: Returns `num_changes`. It is used in `labelChanges` in `MainWindow` class.
- `void sort_bubble(int* arr, int size)`: Bubble sort. Repeatedly steps through the list and compares adjacent elements, and swaps them if they are in wrong order. Repeat the procedure until `arr` is sorted. Note that after n^{th} iteration, `arr[size-n]` is in correct position, so we. don't need to consider them in remaining procedure. This optimization is applied in the algorithm.
- `void sort_insertion(int* arr, int size)`: Insertion sort. Calls `insert(int* arr, int size, int i)` function for each `i` in increasing order.
- `void sort_selection(int* arr, int size)`: Selection sort. Look for the minimum value from the i^{th} element to the last element, and swap the minimum value with i^{th} value, for each `i` in increasing order.
- `void sort_merge(int* arr,int size, int start, int end)`: Merge sort. It is a recursive algorithm. Divide arr into 2 subarrays of smaller size, and use Merge sort to sort those 2 subarrays, then merge them using `merge(int* arr, int size, int start, int end)` function.
- `void sort_heap(int* arr, int size)`: Divide `arr` into sorted part and unsorted part, finds the largest element in unsorted part using heap structure, then move the largest element in unsorted part to sorted part, and repeat the procedure.

- `void sort_quick(int* arr, int size, int start, int end):` Quicksort. Choose a pivot element, and place all elements smaller than pivot to the le ft of pivot, and all the elements larger than pivot to right of pivot. Then use quicksort again for the smaller part and larger part. In our project, rightmost element is always chosen to be pivot.
- `void sort_radix(int* arr, int size, int base):` Radix sort. Us e stable sorting algorithm to sort `arr`, not based on their exact value, but based in the ir value at certain digit when they are represented in number of base base. Repeat the procedure until all the digits are considered. In our project, the sorting order is set to be from least significant digit to most significant digit, and counting sort is used for t he stable sorting algorithm. It is not a comparison-based sorting algorithm, thus it doe s not increase the `num_comparisons`. In our project, base 2 and base 10 are used.
- `void sort_counting(int* arr, int size, int base, int exp):` Counting sort. Use another array to store the frequency of each element `in arr`, a nd use frequency array to find the correct place for each element and put them into `a rr`. It is not a comparison-based sorting algorithm, thus it does not increase the `num_ comparisons`.
- `void sort_gnome(int* arr, int. size):` Gnome sort. Similar to insert ion sort. But instead of using linear search to find the correct relative position, it repe atedly swaps the element with element before that until the element is in correct relati ve position.
- `void sort_cocktail(int* arr, int size):` Cocktail sort (also known as Cocktail shaker sort). Similar to bubble sort, but after each iteration, it iterates in r everse direction too (hence it canbe considered as bi-directional bubble sort). Note th at after n^{th} iteration, `arr[n-1]` and `arr[size-n]` are in correct position, thus w

e do not need to consider them in the sorting procedure. This optimization us applied in this project.

- `void sort_comb(int* arr, int size):` Comb sort. Similar to bubble so rt, but instead of comparing adjacent elements, it compares elements with certain gap between their position and swap them if they are in wrong relative position, then shri nk the gap (unless the gap is 1) by certain factor and repeat the procedure. Initial gap of $\frac{size}{2}$ and shrinking factor of 1.3 is used in this project.
- `void sort_oddeven(int* arr, int size):` Odd Even sort. Similar to b ubble sort, but it first iterate for all `(odd,even)` indices, then iterate for all `(even, odd)` indices, and repeat the procedure.
- `void visualize(int* arr, int size_arr, int* color_index, int color_size):` Update the current state of `arr` to the UI, and colors indices in color_index with different colors, and emit `changed(int* arr, int size _arr, int* index, int size_color)` signal.
- `void changed(int* arr, int size_arr, int* index, int size _color):` A signal to indicate that `arr` is changed. Used in `onNumbersChange d(int*, int, int*, int)` slot in `MainWindow` class.
- `void done():` A signal to indicate that the sorting is done. Used in `onSorting Finished()` slot in `MainWindow` class.

Paint

Paint class is the derived class of `QWidget` class and it is a class to draw elements on the palette. It controls width, space and color of each bar. It is the member variable of MainWindow class.

Member variables explanation

- `bool animation` : true when elements are moving, false when not
- `int penWidth` : the width of each element, it changes according to the number of elements
- `int spacing` : space between each element, default is 0
- `int* numbers` : array of elements
- `int* indices` : array of indices to be painted in different color

- `int size` : size of numbers array, use this when we paint all the elements by iterating through numbers array by for-loop
- `int sizeColorIndices` : size of indices array, use this for iteration
- `QPalette palette` : A QPalette makes the UI easily configurable and easier to keep color consistent. It specifies the background color of the painting zone
- `QPen pen` : QPen is the class to draw lines. Each element is drawn by pen
- `QColor lineColor` : color of each element
- `QColor backgroundColor` : background color of the palette.
- `QString paintType` : The type of element(Bar or Star)
- `QStringList paintTypes` : A Qt style array of string it contains “Bar” and “Star” user can select this from the dropmenu in ui
- `std::vector<QColor> colors` : A vector of QColor, which stores the possible colors of each element. (red or green)
- `QMediaPlayer* sortingsound` : A sound for each element movement

Member functions explanation

- `Paint(QWidget *parent = nullptr)`

  A constructor for Paint class. It sets the background color of the palette(painting area). It also sets the initial values of member variables `animation`, `Penwith`, `spacing` and `pen`. Finally, it initializes `sorting sound` using the binaries of the sound stored in the `Resources` folder.

These are setter functions, which simply set the value except for the function `setPenWidth(int)`. `setPen(const QPen)` and `setBrush(const QBrush)` are using pass by reference in order to save memory.

- `void setPenWidth(int width)` : If `width` is greater than the MAX_PEN_WIDTH it sets the value of constant variable MAX_PEN_WIDTH otherwise sets the value of `width`.
- `void setPen(const QPen &pen)`
- `void setBrush(const QBrush &brush)`
- `void setSpacing(int space)`
- `void setPaintType(QString option)`
- `void setLineColor(QColor color)`
- `void setAnimation(bool anim)`
- `void setPaintData(int*, int*, int, int)` : this function sets member variables `numbers`, `colorIndices`, `size` and `sizeColorIndices` which are essential to draw elements.

- `QStringList getPaintTypes()`

  It simply returns a QStringList of paintTypes which are `Bar` and `Star`.

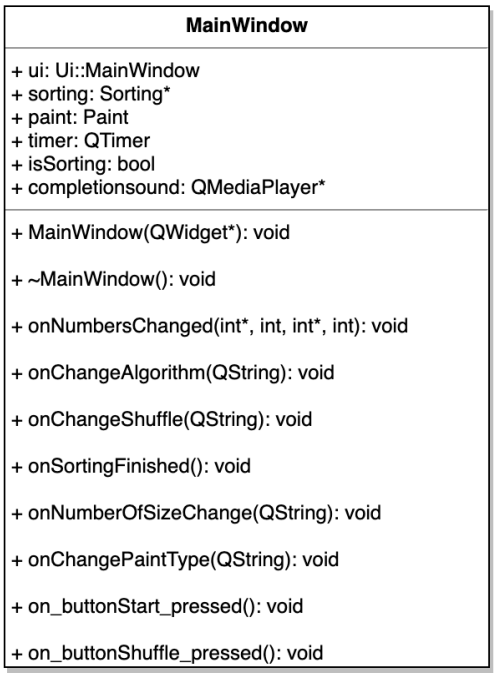
- `void reset()`

  It resets the member variable `linecolor` to constant variable `DEFAULT_LINE_COLOR` which is Qt::lightGray.

- `void paintEvent(QPaintEvent *event)`

A paint event is a request to repaint all or part of a widget. It mainly happens if repaint() or update() is invoked. This is the core function to draw elements. First, we initialize a local variable of QPainter class called painter. Next, we initialize an int variable called `colorIdx` with initial value 0. It also initialize an int variable `space`, which is a space to set the elements in the middle of the palette. It iterates through the `numbers` array and check if the index matches with any of the value in colorIndices it becomes a colored bar. It draws each bar with a function `painter.drawLine(int,int,int,int)` for a bar and a function `painter.drawPoint(int,int)` for a star. Finally, it checks the state of `sortingsound`. If it is in ` QMediaPlayer::PlayingState`, it set the position of `sortingsound` to 0. In `QMediaPlayer::StoppedState`, it simply play `sortingsound`.

MainWindow



MainWindow class is the derived class of `QMainWindow` class and it is the central component of this application. It interacts with all the user-defined classes and connects logic parts and UI parts.

Member variables explanation

- `Ui::MainWindow *ui` : A default variable provided by Qt. Access `mainwindow.ui` via this variable and control UI components.
- `Sorting* sorting` : An instance variable of Sorting class. To work on the sorting algorithm is the main roll.
- `Paint paint` : An instance variable of Paint class. This class is used to visualize the elements.
- `QTime timer` : A timer for measuring execution time of sorting.
- `isSorting` : true when elements have been sorting, false when not

- `QMediaPlayer* completionsound` : A sound for the completion of sorting

Member functions explanation

- `MainWindow(QWidget)`

  A constructor of MainWindow class. To set up the UI components, connect signals and slots and initialize `completionsound` are the main task for this function. It calls `createArray()` function at the end.

- `~MainWindow()`

  A destructor of MainWindow class. It simply deletes `completionsound` and `ui` variables.

- `void onNumbersChanged(int*, int, int*, int)`

  Signal for this slot function is `Sorting::changed()` and it handles the array of numbers and colorIndices with the size of each array. It gets the number of comparisons and changed and use those value to update the UI components `labelComparisons` and `labelChanges`. It calls `paint.setPaintData()` and set the member variables of Paint instance and call `paint.update()` to trigger `paintEvent()`.

These slot functions are connected to a signal function `currentTextChanged(QString)` of UI components `comboBoxAlgorithm`, `comboBoxShuffle` or `comboBoxPaintType`. When user changes the option in comboBox this slot function is called. It calls the appropriate setter function of paint class and set the new value.

- `void onChangeAlgorithm(QString)`
- `void onChangeShuffle(QString)`
- `void onChangePaintType(QString)`

Other slot functions

- `void onSortingFinished()` : This slot function is connected to signal function `Sorting::done()`. This function is called when sorting is finished. It sets text of ui component `labelTime`, change the color of elements to green, update the buttons and make the `completionsound` at the end. The logic of making the sound is same as `sortingsound`.

- `void onNumberOfSizeChange(QString)` : This slot function is to deal with the changes in array size. It is connected to UI component `comboBoxArraySize`. It sets the proper value of penWidth and update the UI.

- `void on_buttonStart_pressed()` : this is called when `buttonStart` is pressed. If `isSorting` is false it sets the `isSorting` true and change the button accordingly. It starts the sorting animation and the timer. If `isSorting` is true it sets the `isSortin` false and it stops sorting(thread) calling `terminate()` which is a function of QThread.

- `void on_buttonShuffle_pressed()` : this is called when buttonShuffle is pressed. It reset the line color of elements and call `Sorting::createArray`.

Data structures used

There are many types of data structures which can represent a list of numbers (e.g. linked list and array). In this project, we use an array (`arr` in `Sorting` class and `numbers` in `Paint` class) to represent the list of numbers. to be sorted, instead of using a linked list. We chose to use an array, instead of a linked list, for this project for numerous reasons:

- Faster & easier access to the numbers. We can directly access to the number at specific index if we use an array. If we use a linked list, we must access the numbers sequentially from the beginning of the list. Since we access element of the list many times throughout the sorting procedure, this greatly increase the performance of the sorting.
- Memory-saving. Using an array to represent a list of numbers uses significantly less memory than using a linked list to represent the same list of numbers because linked list requires extra memory to store pointers.
- Size of the list does not change in the sorting procedure. Although the size of the list is changeable, all the changes are made before the sorting begins, thus using a linked list in unnecessary.
- Simplicity in term of coding.

Also, dynamic array is used instead of static array to allow users to choose different sizes of the array for the visualization. Although the size of the array is fixed during the sorting procedure, users can change the size of the array before the sorting procedure begins. For similar reasons, dynamic array (`color_index` in `Sorting` class and `colorIndices` in `Paint` class) is used to store the indices of the numbers to be represented using different color in the visualization.

Conclusion

To conclude, this project visualizes various sorting algorithms. It also shows the number of comparisons and changes made, which sum of them is a measure of time complexity of the algorithm. This is important because many important algorithms require sorting of the data to be done before the algorithms are actually applied. For example, binary searching algorithm requires a sorted list of data before the actual searching begins, greedy algorithm of interval scheduling also requires jobs to be sorted based on their finish time, Kruskal's algorithm on building minimum spanning tree of a weighted graph also requires edges to be sorted based on their weights. To maximize the performances of these algorithms, choice of suitable sorting algorithm according to the initial list is necessary. This project provides those useful information related to sorting algorithms, and shows users how different sorting algorithms work on different initial arrays (differ in size of the array, or order of elements).