```cpp
#include <QApplication>
#include <QMainWindow>
#include <QWidget>
#include <QVBoxLayout>
#include <QHBoxLayout>
#include <QLabel>
#include <QPushButton>
#include <QLineEdit>
#include <QGraphicsView>
#include <QGraphicsScene>
#include <QGraphicsRectItem>
#include <QTimer>
#include <QRadioButton>
#include <QIntValidator>
#include <QComboBox>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <QSlider>
#include <chrono>

class SortingVisualizer : public QMainWindow
{
    Q_OBJECT

public:
    SortingVisualizer(QWidget* parent = nullptr) : QMainWindow(parent)
    {
        setupUi();
        timer = new QTimer(this);
        connect(timer, &QTimer::timeout, this, &SortingVisualizer::visualizeSortStep);
    }

private slots:
    void generateArray();
    void startVisualization();
    void pauseVisualization();
    void adjustVisualizationSpeed(int value);
    void visualizeSortStep();
    void updateArraySizeFromInput();
    void disableCustomInput(bool checked);
    void enableCustomInput(bool checked);
    void analyzeTimeComplexity();
    void onAlgorithmChanged(int index);
    void visualizeSortingStepByStep(); // New function for step-by-step sorting

private:
    void setupUi();
    void visualizeSorting();
    void updateArraySizeLabel();

    QWidget* centralWidget;
    QVBoxLayout* mainLayout;
    QLabel* arraySizeLabel;
    QLineEdit* arraySizeInput;
```

```
    QPushButton* generateButton;
    QPushButton* startButton;
    QPushButton* pauseButton;
    QSlider* speedSlider;
    QGraphicsView* visualizationView;
    QGraphicsScene* visualizationScene;
    QTimer* timer;
    std::vector<int> arr;
    int currentIndex;
    bool arrayGenerated;
    int sortI, sortJ;
    bool sortSwapped;
    int currentSortStep; // New variable to keep track of the sorting step

    bool quickSortStep(std::vector<int>& arr, int low, int high);
    int quickSortPartition(std::vector<int>& arr, int low, int high);
    bool mergeSortStep(std::vector<int>& arr, int left, int right);

    QRadioButton* randomArrayModeRadioButton;
    QRadioButton* customArrayModeRadioButton;
    QLineEdit* customInputWidget;
    QLabel* executionTimeLabel;
    QPushButton* analyzeButton;
    int heightFactor = 5;
    QComboBox* algorithmComboBox;

    enum SortingAlgorithm {
        BubbleSort,
        InsertionSort,
        SelectionSort,
        QuickSort,
        MergeSort // Add more sorting algorithms here
    };
    SortingAlgorithm currentAlgorithm;

    void bubbleSort(std::vector<int>& arr);
    void insertionSort(std::vector<int>& arr);
    void selectionSort(std::vector<int>& arr);
    void quickSort(std::vector<int>& arr, int low, int high);
    void mergeSort(std::vector<int>& arr, int left, int right);
    void merge(std::vector<int>& arr, int left, int middle, int right);
    void customSwap(int i, int j);

    void runSortingAlgorithm();
};

void SortingVisualizer::setupUi()
{
    centralWidget = new QWidget(this);
    setCentralWidget(centralWidget);

    mainLayout = new QVBoxLayout(centralWidget);

    // Algorithm Selection
    algorithmComboBox = new QComboBox(this);
```

```
    algorithmComboBox->addItem("Bubble Sort");
    algorithmComboBox->addItem("Insertion Sort");
    algorithmComboBox->addItem("Selection Sort");
    algorithmComboBox->addItem("Quick Sort");
    algorithmComboBox->addItem("Merge Sort");
    mainLayout->addWidget(algorithmComboBox);

    // First Line: Array Size Label and Input
    QHBoxLayout* arraySizeLayout = new QHBoxLayout;
    arraySizeLabel = new QLabel("Array Size:", this);
    arraySizeLayout->addWidget(arraySizeLabel);
    arraySizeInput = new QLineEdit(this);
    arraySizeInput->setPlaceholderText("Enter the array size");
    arraySizeInput->setValidator(new QIntValidator(5, 100, this));
    arraySizeLayout->addWidget(arraySizeInput);
    mainLayout->addLayout(arraySizeLayout);

    // Second Line: Random Array and Custom Array Options
    QHBoxLayout* arrayOptionsLayout = new QHBoxLayout;
    randomArrayModeRadioButton = new QRadioButton("Random Array", this);
    arrayOptionsLayout->addWidget(randomArrayModeRadioButton);
    customArrayModeRadioButton = new QRadioButton("Custom Array", this);
    arrayOptionsLayout->addWidget(customArrayModeRadioButton);
    mainLayout->addLayout(arrayOptionsLayout);

    // Third Line: Custom Array Element Input
    customInputWidget = new QLineEdit(this);
    customInputWidget->setPlaceholderText("Enter custom array elements (comma-
separated)");
    mainLayout->addWidget(customInputWidget);

    // Fourth Line: Generate Array Button
    generateButton = new QPushButton("Generate Array", this);
    mainLayout->addWidget(generateButton);

    // Fifth Line: Start Visualization Button
    startButton = new QPushButton("Start Visualization", this);
    mainLayout->addWidget(startButton);

    // Sixth Line: Pause Visualization Button
    pauseButton = new QPushButton("Pause Visualization", this);
    pauseButton->setEnabled(false);
    mainLayout->addWidget(pauseButton);

    // Seventh Line: Speed Slider
    speedSlider = new QSlider(Qt::Horizontal, this);
    speedSlider->setMinimum(1);
    speedSlider->setMaximum(100);
    speedSlider->setValue(50);
    mainLayout->addWidget(speedSlider);

    // Eighth Line: Visualization Area
    visualizationView = new QGraphicsView(this);
    visualizationScene = new QGraphicsScene(this);
    visualizationView->setScene(visualizationScene);
```

```
    mainLayout->addWidget(visualizationView);

    // Ninth Line: Execution Time Label
    executionTimeLabel = new QLabel("Execution Time: N/A ms", this);
    mainLayout->addWidget(executionTimeLabel);

    // Tenth Line: Analyze Button
    analyzeButton = new QPushButton("Analyze Time Complexity", this);
    mainLayout->addWidget(analyzeButton);

    // Connect the button click to a slot for analysis
    connect(analyzeButton, &QPushButton::clicked, this,
&SortingVisualizer::analyzeTimeComplexity);
    connect(algorithmComboBox, QOverload<int>::of(&QComboBox::currentIndexChanged),
this, &SortingVisualizer::onAlgorithmChanged);

    connect(generateButton, &QPushButton::clicked, this,
&SortingVisualizer::generateArray);
    connect(startButton, &QPushButton::clicked, this,
&SortingVisualizer::startVisualization);
    connect(pauseButton, &QPushButton::clicked, this,
&SortingVisualizer::pauseVisualization);
    connect(arraySizeInput, &QLineEdit::editingFinished, this,
&SortingVisualizer::updateArraySizeFromInput);
    connect(randomArrayModeRadioButton, &QRadioButton::clicked, this,
&SortingVisualizer::disableCustomInput);
    connect(customArrayModeRadioButton, &QRadioButton::clicked, this,
&SortingVisualizer::enableCustomInput);
    connect(speedSlider, &QSlider::valueChanged, this,
&SortingVisualizer::adjustVisualizationSpeed);

    arrayGenerated = false;
    sortI = sortJ = -1;
    sortSwapped = false;
    currentSortStep = 0; // Initialize the sorting step
    currentAlgorithm = BubbleSort;

    onAlgorithmChanged(algorithmComboBox->currentIndex());
}

void SortingVisualizer::generateArray()
{
    bool isRandomArrayMode = randomArrayModeRadioButton->isChecked();
    bool isCustomArrayMode = customArrayModeRadioButton->isChecked();

    if (isRandomArrayMode)
    {
        int size = arraySizeInput->text().toInt();
        arr.clear();

        srand(static_cast<unsigned>(time(nullptr)));
        for (int i = 0; i < size; ++i)
        {
            arr.push_back(rand() % 100 + 1);
        }
```

```
        customInputWidget->setEnabled(false);
    }
    else if (isCustomArrayMode)
    {
        customInputWidget->setEnabled(true);

        QString input = customInputWidget->text();
        QStringList inputList = input.split(",", Qt::SkipEmptyParts);
        arr.clear();

        for (const QString& str : inputList)
        {
            bool ok;
            int value = str.toInt(&ok);
            if (ok)
            {
                arr.push_back(value);
            }
        }
    }

    arrayGenerated = true;
    updateArraySizeLabel();
    visualizeSorting();
    adjustVisualizationSpeed(speedSlider->value());
}

void SortingVisualizer::startVisualization()
{
    startButton->setEnabled(false);
    pauseButton->setEnabled(true);
    generateButton->setEnabled(false);
    arraySizeInput->setEnabled(false);
    speedSlider->setEnabled(false);

    // Clear the visualization before starting
    visualizationScene->clear();

    // Reset the sorting variables
    sortI = sortJ = -1;
    sortSwapped = false;

    // Set a reasonable interval (e.g., 100 milliseconds) for the timer
    timer->setInterval(100); // Adjust this value as needed

    // Start the timer for step-by-step visualization
    timer->start();

    // Also, disable the "Random Array" and "Custom Array" options
    randomArrayModeRadioButton->setEnabled(false);
    customArrayModeRadioButton->setEnabled(false);
    customInputWidget->setEnabled(false);

    // Call the new step-by-step sorting function
```

```
    visualizeSortingStepByStep();
}

void SortingVisualizer::pauseVisualization()
{
    startButton->setEnabled(true);
    pauseButton->setEnabled(false);
    generateButton->setEnabled(true);
    arraySizeInput->setEnabled(true);
    speedSlider->setEnabled(true);

    timer->stop();
}

void SortingVisualizer::adjustVisualizationSpeed(int value)
{
    int delay = 101 - value;
    timer->setInterval(delay);
}

void SortingVisualizer::visualizeSorting()
{
    visualizationScene->clear();

    int barWidth = 20;
    int spacing = 5;

    for (int i = 0; i < arr.size(); ++i)
    {
        int rectHeight = arr[i] * heightFactor;
        int yPos = visualizationView->height() - rectHeight;

        QGraphicsRectItem* rect = new QGraphicsRectItem(i * (barWidth + spacing), yPos,
barWidth, rectHeight);
        visualizationScene->addItem(rect);

        QGraphicsTextItem* label = new QGraphicsTextItem(QString::number(arr[i]));
        label->setPos(i * (barWidth + spacing) + barWidth / 2 - 8, yPos - 20);
        visualizationScene->addItem(label);
    }
}

void SortingVisualizer::visualizeSortStep()
{
    // This function is no longer used for step-by-step sorting
}

bool SortingVisualizer::quickSortStep(std::vector<int>& arr, int low, int high)
{
    // Implement Quick Sort step here
}

int SortingVisualizer::quickSortPartition(std::vector<int>& arr, int low, int high)
{
    // Implement Quick Sort partition here
```

```
}

bool SortingVisualizer::mergeSortStep(std::vector<int>& arr, int left, int right)
{
    // Implement Merge Sort step here
}

void SortingVisualizer::merge(std::vector<int>& arr, int left, int middle, int right)
{
    // Implement Merge Sort merge function here
}

void SortingVisualizer::updateArraySizeFromInput()
{
    if (arrayGenerated) {
        updateArraySizeLabel();
        visualizeSorting();
        adjustVisualizationSpeed(speedSlider->value());
    }
}

void SortingVisualizer::updateArraySizeLabel()
{
    arraySizeLabel->setText("Array Size: " + QString::number(arr.size()));
}

void SortingVisualizer::disableCustomInput(bool checked)
{
    if (checked)
    {
        customInputWidget->setEnabled(false);
    }
}

void SortingVisualizer::enableCustomInput(bool checked)
{
    if (checked)
    {
        customInputWidget->setEnabled(true);
    }
}

void SortingVisualizer::runSortingAlgorithm()
{
    if (currentAlgorithm == BubbleSort)
    {
        bubbleSort(arr);
    }
    else if (currentAlgorithm == InsertionSort)
    {
        insertionSort(arr);
    }
    else if (currentAlgorithm == SelectionSort)
    {
        selectionSort(arr);
```

```
    }
    else if (currentAlgorithm == QuickSort)
    {
        quickSort(arr, 0, arr.size() - 1);
    }
    else if (currentAlgorithm == MergeSort)
    {
        mergeSort(arr, 0, arr.size() - 1);
    }

    visualizeSorting();
    analyzeTimeComplexity();
}

void SortingVisualizer::onAlgorithmChanged(int index)
{
    currentAlgorithm = static_cast<SortingAlgorithm>(index);
    if (arrayGenerated) {
        visualizeSorting();
        adjustVisualizationSpeed(speedSlider->value());
    }

    // Add this line to reset the sorting variables
    sortI = sortJ = -1;
    sortSwapped = false;
}


void SortingVisualizer::bubbleSort(std::vector<int>& arr)
{
    int n = arr.size();
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                customSwap(j, j + 1);
            }
        }
    }
}

void SortingVisualizer::insertionSort(std::vector<int>& arr)
{
    int n = arr.size();
    for (int i = 1; i < n; i++)
    {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j--;
        }
```

```
        arr[j + 1] = key;
    }
}

void SortingVisualizer::selectionSort(std::vector<int>& arr)
{
    int n = arr.size();
    for (int i = 0; i < n - 1; i++)
    {
        int minIndex = i;
        for (int j = i + 1; j < n; j++)
        {
            if (arr[j] < arr[minIndex])
            {
                minIndex = j;
            }
        }
        if (minIndex != i)
        {
            customSwap(i, minIndex);
        }
    }
}

void SortingVisualizer::quickSort(std::vector<int>& arr, int low, int high)
{
    // Implement Quick Sort algorithm here
}

void SortingVisualizer::mergeSort(std::vector<int>& arr, int left, int right)
{
    // Implement Merge Sort algorithm here
}

void SortingVisualizer::customSwap(int i, int j)
{
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
    sortI = i;
    sortJ = j;
    sortSwapped = true;
}

void SortingVisualizer::visualizeSortingStepByStep()
{
    if (currentSortStep >= arr.size()) {
        // Sorting is complete
        timer->stop();
        startButton->setEnabled(true);
        pauseButton->setEnabled(false);
        generateButton->setEnabled(true);
        arraySizeInput->setEnabled(true);
        speedSlider->setEnabled(true);
        randomArrayModeRadioButton->setEnabled(true);
```

```
        customArrayModeRadioButton->setEnabled(true);
        customInputWidget->setEnabled(randomArrayModeRadioButton->isChecked());

        // Update the execution time label
        analyzeTimeComplexity();
        return;
    }

    if (sortI >= 0 && sortJ >= 0) {
        // Restore the color of previously compared elements
        visualizationScene->items()[sortI]->setBrush(QColor(Qt::blue));
        visualizationScene->items()[sortJ]->setBrush(QColor(Qt::blue));
    }

    if (sortSwapped) {
        // Swap the elements in the visualization
        visualizeSorting();
        sortSwapped = false;
    }

    // Highlight the current element being processed
    visualizationScene->items()[currentSortStep]->setBrush(QColor(Qt::red));

    // Increment the sorting step
    currentSortStep++;

    // Adjust the visualization speed based on the slider value
    adjustVisualizationSpeed(speedSlider->value());
}

void SortingVisualizer::analyzeTimeComplexity()
{
    if (!arrayGenerated)
        return;

    std::vector<int> copyArr = arr; // Create a copy of the array to avoid modifying
the original

    // Measure the execution time of the sorting algorithm
    auto start = std::chrono::high_resolution_clock::now();
    runSortingAlgorithm();
    auto end = std::chrono::high_resolution_clock::now();

    // Calculate the execution time in milliseconds
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
    long long executionTime = duration.count();

    executionTimeLabel->setText("Execution Time: " + QString::number(executionTime) + "
ms");
}

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
```

```
    SortingVisualizer sortingVisualizer;
    sortingVisualizer.setWindowTitle("Sorting Algorithm Visualizer");
    sortingVisualizer.setGeometry(100, 100, 800, 600);
    sortingVisualizer.show();

    return app.exec();
}
```