UNIVERSITY OF JAMMU
Department of Computer Science & IT

Sorting Benchmarking Tool

A Comparative Study and Implementation in Python and C++

Master of Computer Applications (MCA)
Session: 2025 – 2026

Under the Supervision of
Prof. Vibhakar Mansotra

Head, Department of
Computer Science & IT
University of Jammu

Submitted by:

1. **Anjanee Gouria** ***(0001MCA25)***
2. **Danishwer** ***(0026MCA25)***
3. **Sahil Akhtar** ***(0046MCA25)***

Problem statement

To develop an GUI based application for sorting large sets of number and should have following features: -

1. It should include all the sorting algorithms like BUBBLE SORT, INSERTION SORT, SELECTION SORT, MERGE SORT, QUICK SORT and other algorithm which you find equally good.

2. The input to the application should be given through a file and the size of the initial input should not be less than 10,000 items and this size should increase by the increment of at least 25,000 items and should go up to any extent till system stops responding. The number in the file are to be generated using random number generator.

3. The output after the sorting should also be taken into a file.

4. Every algorithm and every size of number the time taken for sorting should be computed and plot a graph X vs Y (X denotes size and Y denotes time).

5. A complete analysis for each and every algorithm and for size of set of number has to be done for looking into advantage and disadvantage of the algorithm.

Approach to Solve

The main goal of this project was to design and implement a benchmarking tool to test and compare the performance of different sorting algorithms under controlled conditions. The tool should not only execute various algorithms on different datasets but also automatically collect and record performance data for analysis.

To achieve this, the project was divided into multiple clear stages — each focusing on one part of the system, from algorithm design to data collection and analysis.

---

1. Defining the Problem

Sorting is one of the most fundamental operations in computer science, and different sorting algorithms behave differently depending on input size, data distribution, and implementation details.
The problem identified was that theoretical complexities (like $O(n \log n)$ or $O(n^2)$) often fail to represent real-world performance — which depends heavily on factors like hardware speed, compiler optimization, and memory usage.

Hence, this project aims to **measure and compare actual runtime performance** of different sorting algorithms under the same conditions.

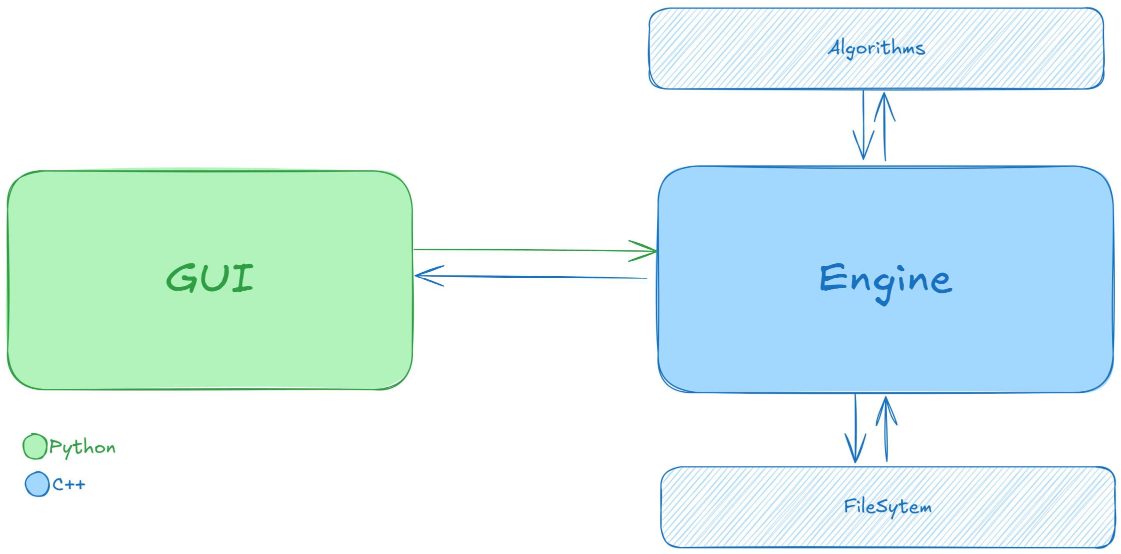
---

2. Designing the Architecture

The overall architecture of the project is divided into four core modules:

1. **GUI Module** — Handles user interaction and visualization.
2. **Engine Module** — Executes sorting operations and performance benchmarking.
3. **File Manager Module** — Manages data input, output, and result storage.
4. **Sorting Module** — Contains all sorting algorithm implementations used for comparison.

High-Level System Structure Diagram:



Choice of Programming Language

C++ was chosen as the primary language for implementing the benchmarking engine because of its high performance, fine-grained memory control, and close-to-hardware execution speed. Since the project's goal was to measure precise algorithmic runtimes, it was essential to minimize overhead introduced by language-level abstractions.

Python, while easier for prototyping and GUI development, introduces additional interpretation and garbage collection delays that can distort time measurements. Therefore, the computationally intensive sorting operations were implemented in C++, while the GUI and visualization components were handled through a higher-level interface for better usability and clarity.

3. Implementation Phases

The implementation was carried out in multiple phases:

Phase 1 – Core Algorithm Implementation

All sorting algorithms were implemented using the `std::vector<int>` structure to store and process arrays. Each algorithm was coded in its own class, maintaining clean and reusable code.

Phase 2 – Benchmarking Logic

A benchmarking system was developed to:

- Generate random datasets within a specified numeric range.
- Run each algorithm on the same dataset.
- Measure execution time using high-resolution timers.
- Save the results to a text file (`result.txt`) for later analysis.

Phase 3 – GUI Design

A simple graphical user interface (GUI) was built to improve usability.
The GUI allows the user to:

- Select single or multiple test modes.
- Choose dataset size and value range.
- Start benchmarking and view results in an organized format.

4. Testing Modes

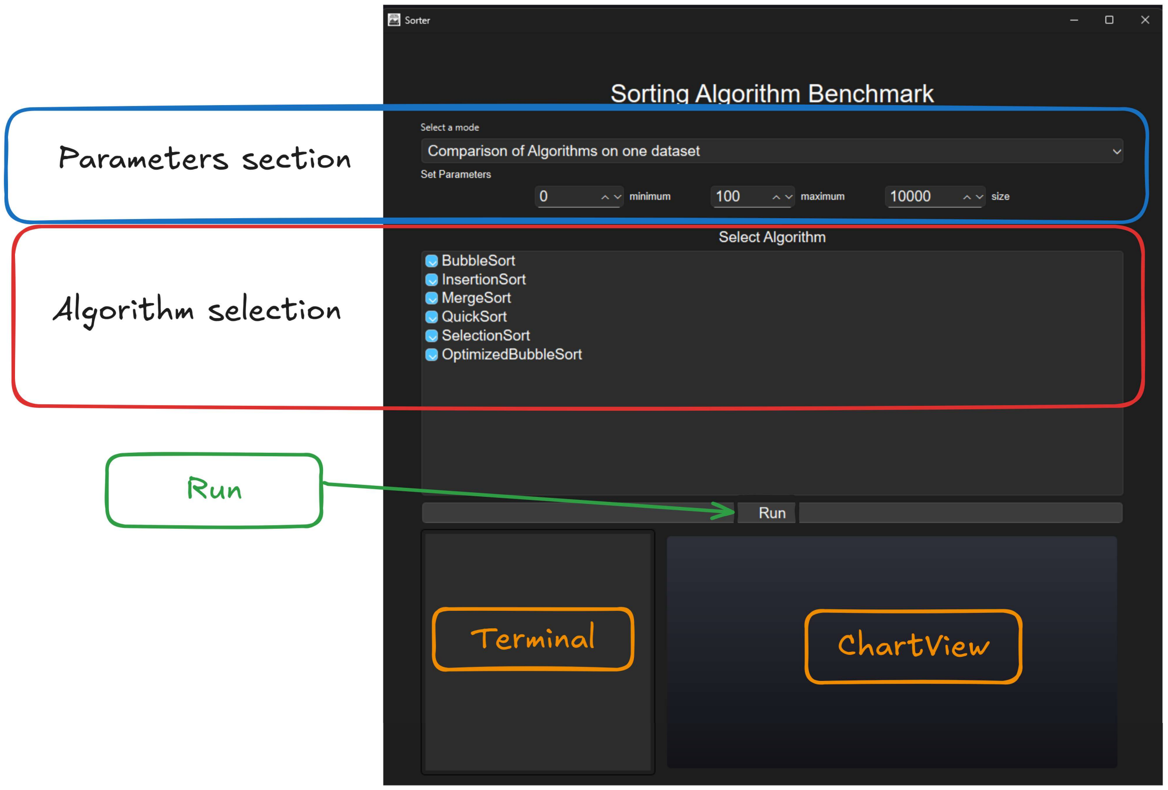
The application supports **two modes of operation**:

Single Dataset Mode

Runs all algorithms once on a single dataset of chosen size and value range.
It is useful for small-scale tests or for demonstrating algorithm behavior visually.

Multiple Dataset Mode

Automatically runs multiple test rounds with increasing dataset sizes.
This mode is ideal for scalability testing, as it shows how each algorithm’s performance changes as input grows larger.



Select a mode
Comparison of Algorithms on one dataset
Set Parameters
0 minimum 100 maximum 10000 size

GUI-Breakdown

5. Data Collection and Storage

After each test, results (including dataset size, algorithm name, and execution time) are written to a file named **result.txt**. This ensures all test outcomes are preserved for later review or statistical analysis.
Each test generates multiple lines of output — making the total dataset quite large, which provides a strong base for reliable conclusions.

---

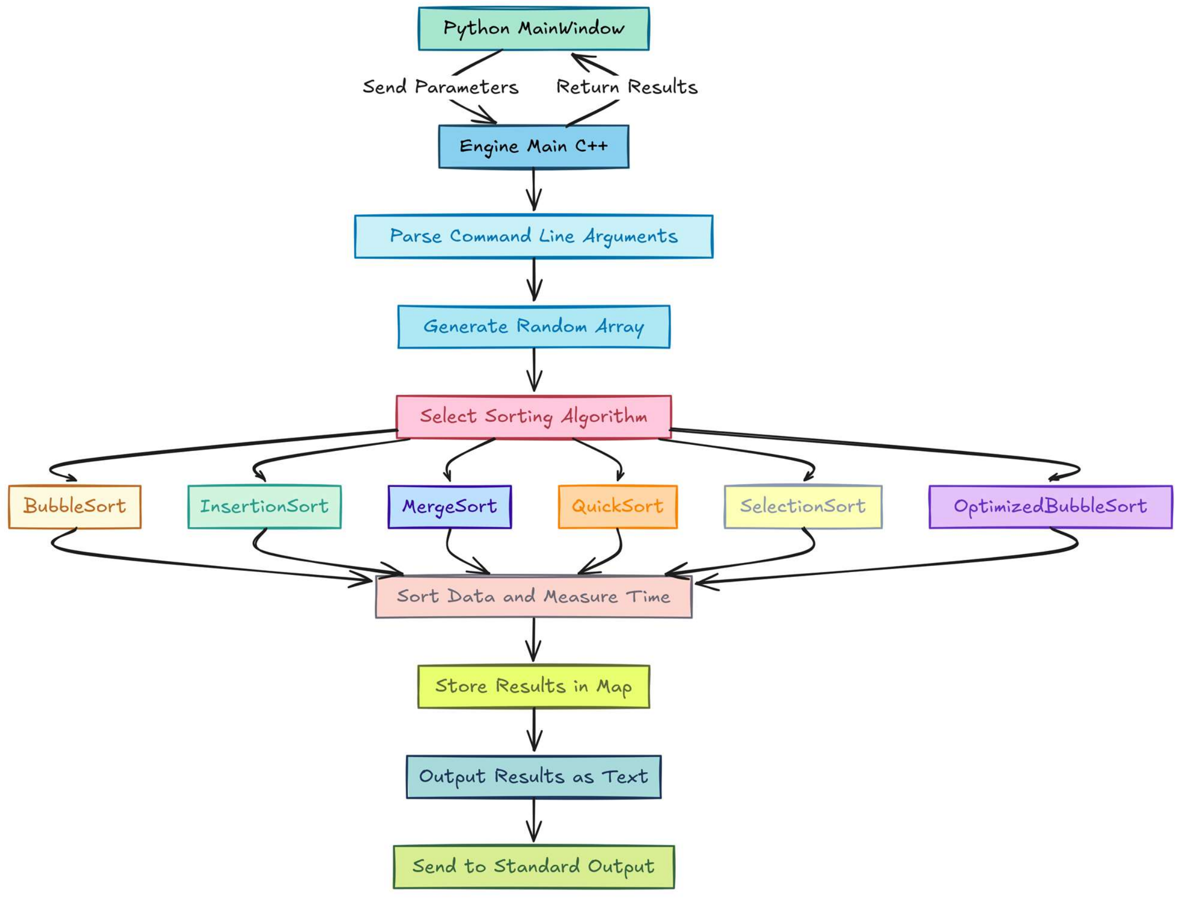
6. Performance Analysis

The recorded data was later processed to:

- Compare the relative performance of algorithms.
- Identify scalability trends.
- Observe how sorting time varies across different input distributions (random vs nearly sorted).

The project also included targeted experiments between selected algorithms (e.g., **Merge Sort vs Insertion Sort**, and **Merge Sort vs Quick Sort**) to analyze how each performs under specific conditions.

Detailed System Structure Diagram



1.Algorithms implemented

The **Sorting Benchmarking Tool** implements six classical sorting algorithms using **C++** object-oriented principles. Each algorithm inherits from a common abstract base class `SortingAlgorithm`, which defines the interface for sorting through a pure virtual function `sort()`.

1. Base Class: SortingAlgorithm

The `SortingAlgorithm` class serves as the parent for all sorting algorithms.
It defines:

- A `name` attribute to store the algorithm's identifier.
- A pure virtual function `sort(vector<int> &arr)` that must be implemented by all derived classes.

This approach ensures a consistent interface and enables polymorphism, allowing the benchmarking engine to call any sorting method dynamically.

1.1 Bubble Sort

Class Name: `BubbleSort`
Description:
Implements the basic Bubble Sort algorithm. It repeatedly traverses the array, comparing adjacent elements and swapping them if they are in the wrong order. The largest element "bubbles" to the end after each pass.

Time Complexity:

- Best: $O(n)$
- Average: $O(n^2)$
- Worst: $O(n^2)$

Space Complexity: $O(1)$

1.2 Optimized Bubble Sort

Class Name: `OptimizedBubbleSort`
Description:
An enhanced version of Bubble Sort that introduces a `swapped` flag

to detect if any swaps occurred during a pass. If no swaps occur, the algorithm terminates early, improving efficiency for nearly sorted lists.

Time Complexity:

- Best: O(n)
- Average: $O(n^2)$
- Worst: $O(n^2)$

Space Complexity: O(1)

1.3 Insertion Sort

Class Name: `InsertionSort`
Description:
Builds the final sorted array one element at a time. It picks elements from the unsorted portion and places them in the correct position within the sorted portion.

Time Complexity:

- Best: O(n)
- Average: $O(n^2)$
- Worst: $O(n^2)$

Space Complexity: O(1)

1.4 Selection Sort

Class Name: `SelectionSort`
Description:
Finds the smallest element from the unsorted array and swaps it with the first unsorted element. This process repeats until the entire array is sorted.

Time Complexity:

- Best: $O(n^2)$
- Average: $O(n^2)$
- Worst: $O(n^2)$

Space Complexity: O(1)

1.5 Merge Sort

Class Name: `MergeSort`
Description:
A divide-and-conquer algorithm that recursively divides the array into halves, sorts each half, and merges the sorted halves into a single sorted array.

Time Complexity:

- Best: $O(n \log n)$
- Average: $O(n \log n)$
- Worst: $O(n \log n)$

Space Complexity: $O(n)$

1.6 Quick Sort

Class Name: `QuickSort`
Description:
Also a divide-and-conquer algorithm. It selects a pivot element, partitions the array into two subarrays (elements smaller and greater than the pivot), and recursively sorts the subarrays.

Time Complexity:

- Best: $O(n \log n)$
- Average: $O(n \log n)$
- Worst: $O(n^2)$

Space Complexity: $O(\log n)$

2. Mathematical Model

Let the system be represented as a **tuple:**

$$S = \{I, O, F, C, P\}$$

Where:

- **I** : Set of inputs
- **O** : Set of outputs
- **F** : Set of functions (operations/processes)
- **C** : Set of constraints
- **P** : Performance parameters

2.1 Input (I)

$$I = \{min, max, n, alg\}$$

Where:

- **min,max** → limit of random numbers
- **n** → Size of array (|A|)
- **alg** → Selected sorting algorithm from the set

$$alg \in \{BubbleSort, OptimizedBubbleSort, InsertionSort, SelectionSort, MergeSort, QuickSort\}$$

2.2 Output (O)

$$O = \{A', t\}$$

Where:

- **A'** → Sorted array (ascending order)
- **t** → Time taken by algorithm (in milliseconds)

2.3 Functional Mapping (F)

The benchmark engine performs a mapping between input and output:

$$F: I \rightarrow O$$
$$F(A, n, alg) = \{sort_{alg}(A), time(sort_{alg})\}$$

That is – for each algorithm, it returns the sorted array and the execution time.

2.4 Constraints (C)

$$C = \{n > 0,\ A_i \in \mathbb{Z}, A_i > min, A_i < max,\ alg \in ALGORITHMS\}$$

- Input size must be positive
- Array elements must be integers and in the range min-max.
- Algorithm must be one of the defined six sorting algorithms

2.5 Performance Parameters (P)

$$P = \{T, S\}$$

Where:

- **T** → Time complexity of the selected algorithm
- **S** → Space complexity of the selected algorithm

3.Low Level code documentation

- GUI Module

Purpose

This file defines the graphical user interface (GUI) for the Sorting Algorithm Benchmark application using PySide6.
It allows users to select sorting algorithms, configure parameters, run benchmarks through the C++ engine, and view results both in textual and graphical form.

Class: `MainWindow(QMainWindow)`

Overview

The `MainWindow` class serves as the primary GUI window. It manages layout construction, user interactions, and communication with the external C++ engine process.

Key Attributes

- **mode** – Stores the current mode (`SINGLE_MODE` or `MULTIPLE_MODE`).
- **result** – Holds benchmark results in multiple-run mode.
- **chart** – QChart object for displaying graphical results.
- **terminal** – Displays real-time textual output from the engine.
- **progress** – Visual progress bar during engine execution.
- **select_mode** – Dropdown for selecting benchmark mode.
- **select** – List widget for selecting sorting algorithms.

Functional Overview

1. Mode Selection

The application supports two modes:

- **Single Mode**: Runs the benchmark for one input size.
- **Multiple Mode**: Runs the benchmark across several input sizes.

When the mode is changed using the combo box, the UI is re-rendered accordingly:

```
self.select_mode.currentIndexChanged.connect(self.mode_changed)
```

---

2. Parameter Configuration

The interface allows the user to set:

- **Minimum value** of array elements (`self.min`)
- **Maximum value** of array elements (`self.max`)
- **Array size** (single integer or multiple values)

If multiple mode is active, array sizes can be entered as comma-separated values.

---

3. Algorithm Selection

The available sorting algorithms are loaded dynamically into a checklist:

```
for algo in ALGORITHMS:
    item = QListWidgetItem(algo)
    item.setCheckState(Qt.Checked)
```

Algorithms included:

- BubbleSort
- InsertionSort
- MergeSort
- QuickSort
- SelectionSort
- OptimizedBubbleSort

---

4. Engine Execution

The application launches the external C++ engine using `QProcess`.

Single Mode:

```
self.run_engine(min, max, size, algo_list)
```

Multiple Mode:

```
self.run_engine_Mutiple(min, max, size, algo_list)
```

Both methods run the executable located at `ENGINE_PATH` and capture its standard output.

---

5. Process Handling

Output Reading:
The method `process_output()` continuously reads engine output and updates the terminal and progress bar.

```
process.readyReadStandardOutput.connect(lambda: self.process_output(process))
```

Completion Handling:
After each process finishes, `process_finished()` parses and displays results.
In single mode, a sorted summary of runtimes is displayed.
In multiple mode, results are accumulated across input sizes and visualized after all runs are completed.

---

6. Chart Visualization

Two types of visualizations are supported:

Single Mode Chart:
Displays a bar chart comparing runtime of algorithms for a single array size.

```
set_chart_single(results)
```

Multiple Mode Chart:
Displays a line chart showing performance across various array sizes.

```
set_chart_mutiple(results)
```

Both utilize `QChart`, `QBarSeries`, `QLineSeries`, and axis classes for rendering.

---

7. Helper Functions

- **get_label()** – Creates a styled `QLabel` widget.
- **get_box()** – Combines multiple widgets into a horizontal layout.
- **get_selected()** – Returns the list of selected algorithms.
- **parse_results()** – Parses the output text received from the C++ engine.

– Engine Module

1. Class: Engine

Purpose:
Handles execution of sorting algorithms, timing, and file operations.
It acts as the central controller for reading data, invoking sorting, and writing results.

Attributes:

- `File f` – Represents the input/output file.
- `SortingAlgorithm *algo` – Pointer to the sorting algorithm used.
- `static const char delimiter` – Defines newline character as data separator.

Methods:

- `int run()` – Reads numbers from file, sorts them, measures execution time, and writes the sorted data back.
- `vector<int>& string_to_vector(string str)` – Converts file contents into integer vector.
- `static string vector_to_string(vector<int> arr)` – Converts sorted vector back to string format.
- `static string map_to_string(map<string, int> map)` – Converts performance results into a readable string format.

---

2. Class: RandomNumberGenerator

Purpose:
Generates random integers within a given range, used for preparing test datasets.

Method:

- `static int generate(int min, int max)` – Returns a random integer between `min` and `max`.

---

3. Class: FileFiller

Purpose:
Prepares data files for testing by filling them with randomly generated numbers.

Method:

- `static void fill_with_random(int min, int max, int size, vector<File> files)` –
  Generates `size` random numbers and writes the same dataset into multiple files for benchmarking.

---

4. Class: Tester

Purpose:
Coordinates the testing of multiple sorting algorithms.
Creates files, fills them with random data, runs sorting via the `Engine`, and stores the execution times.

Method:

- `static map<string, int>& test(vector<SortingAlgorithm*> algorithms, int min, int max, int size)` –
  Executes each algorithm on identical datasets and collects benchmark results.

---

5. Function: main()

Purpose:
Entry point of the program.
Parses command-line arguments, initializes algorithms, triggers testing, and stores results.

Flow:

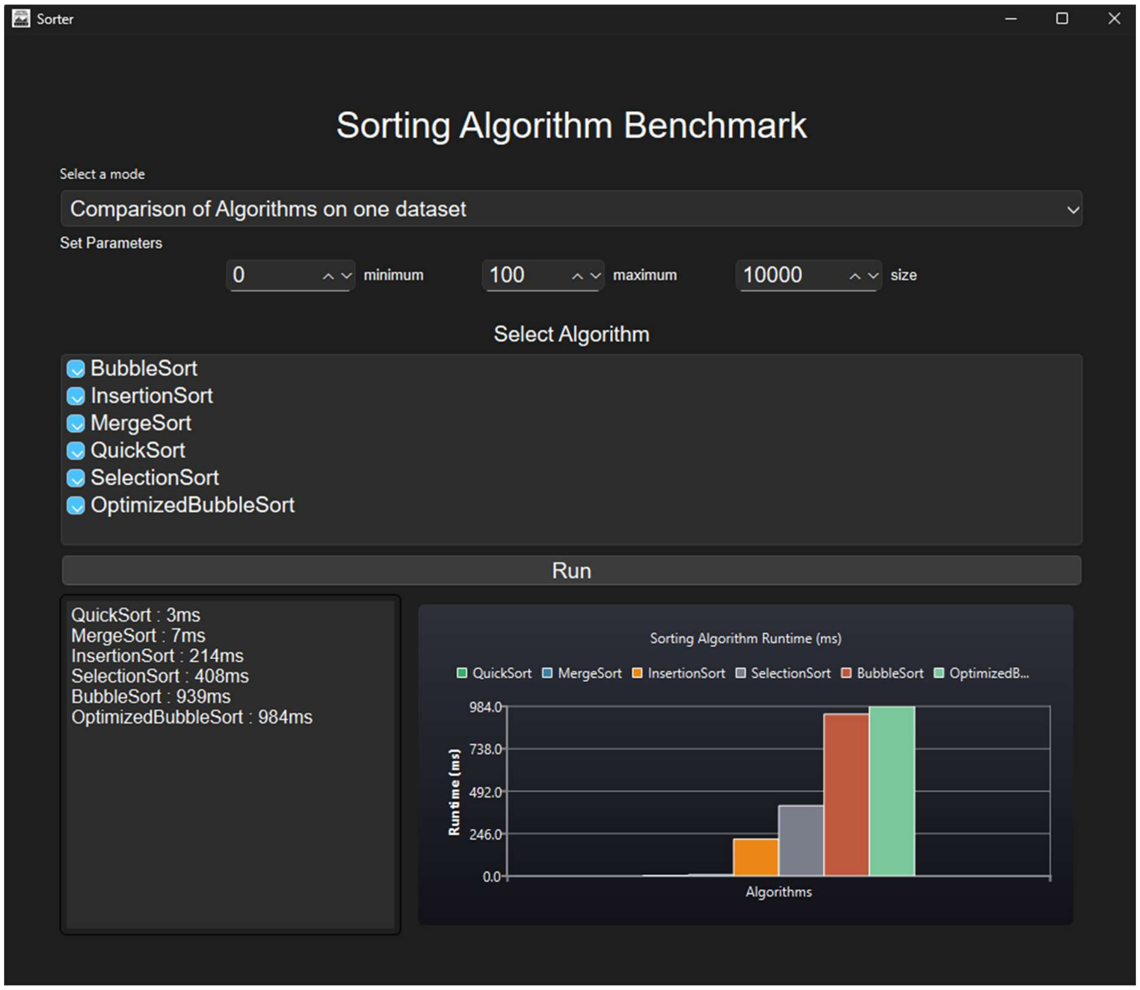
1. Read `min`, `max`, and `size` from arguments.
2. Load selected algorithms (or defaults if none provided).
3. Call `Tester::test()` to run benchmarks.
4. Save final results to `results.txt`.

---

6. Dependencies

- **`file_manager.h`** : Manages reading and writing to files.
- **`sorting_algorithm.h`** : Defines sorting algorithm classes
- **`<chrono>`** → Used for timing execution durations.
- **`<map>`**, **`<vector>`**, **`<string>`** → STL containers for data management.

4.Graphical User Interface (GUI)



```
QuickSort : 3ms
MergeSort : 7ms
InsertionSort : 214ms
SelectionSort : 408ms
BubbleSort : 939ms
OptimizedBubbleSort : 984ms
```

The graphical interface of the *Sorting Benchmarking Tool* is developed using **PySide6 (Qt for Python)**.
It provides an intuitive layout where the user can select sorting algorithms, define data range and size, and choose between single or multiple comparison modes. Once the parameters are set, pressing the **Run** button initiates benchmarking, displaying real-time results in both **text format** and **interactive charts**.
The interface dynamically updates execution progress, visualizes runtime comparisons for each algorithm, and ensures a smooth user experience through a clean, dark-themed design.

5.Source Code

---------------------------main.py---------------------------

```
from PySide6.QtWidgets import *
from PySide6.QtGui import *
from PySide6.QtCharts import *
from PySide6.QtCore import *
import sys
from const import *
from collections import defaultdict

DEFAULT_FONT = QFont("Arial", 14)


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Sorter")
        self.setGeometry(0, 0, 1000, 1000)
        self.setWindowIcon(QIcon("./GUI_python/icon.png"))
        self.result = []
        self.count = 0

        #! mode selector
        self.mode = SINGLE_MODE
        self.mode_label = QLabel("Select a mode", self)
        self.select_mode = QComboBox()
        self.select_mode.setFont(DEFAULT_FONT)
        self.select_mode.setStyleSheet("padding:4px;")
        self.select_mode.addItems(MODE_NAME)
        self.select_mode.currentIndexChanged.connect(self.mode_chang
ed)

        #! drawing UI
        self.draw_UI()

    def mode_changed(self):

        self.mode = MODE_VALUE[self.select_mode.currentText()]
        #!redrawing UI
        self.draw_UI()

    def draw_UI(self):

        #! central widget and layout
        centre = QWidget()
        self.setCentralWidget(centre)

        #! title
        label = QLabel("Sorting Algorithm Benchmark", self)
        label.setFont(QFont("Arial", 24))
        label.setAlignment(Qt.AlignCenter)
        label.setStyleSheet("padding:10px;")

        #! setting parameters (min, max , size)
        parameter_info = self.get_label(
            "Set Parameters", font=QFont("Arial", 10),
align=Qt.AlignLeft)

        #!Min
        l_min = self.get_label("minimum", font=QFont("Arial", 10))
        self.min = self.get_range_input(MIN_LIMIT, 0, 0)
        min_box = self.get_box(l_min, self.min)
        #!Max
        l_max = self.get_label("maximum", font=QFont("Arial", 10))
        self.max = self.get_range_input(0, MAX_LIMIT, 100)
        max_box = self.get_box(l_max, self.max)
        #!Size
        l_size = self.get_label("size", font=QFont("Arial", 10))

        #! if mode is SINGLE
        if (self.mode == SINGLE_MODE):
            self._size = self.get_range_input(1, SIZE_LIMIT, 10000)

        #! if mode is MULTIPLE
        if (self.mode == MULTIPLE_MODE):
            self._size = QTextEdit()
            self._size.setFont(DEFAULT_FONT)
            self._size.setMaximumHeight(30)
            self._size.setText(DEFAULT_SIZES)
        self_box = self.get_box(l_size, self._size)

        #! select algorithm
        l2 = self.get_label("Select Algorithm")

        #! algo selector
        self.select = QListWidget()
        self.select.setFont(DEFAULT_FONT)
        self.select.setMinimumHeight(150)

        #! adding algorithms
        for algo in ALGORITHMS:
            item = QListWidgetItem(algo)
```

```
                item.setFlags(Qt.ItemIsUserCheckable | Qt.ItemIsEnabled)
                item.setCheckState(Qt.CheckState.Checked)
                self.select.addItem(item)

        #! run
        self.button = QPushButton("Run", self)
        self.button.setFont(DEFAULT_FONT)
        self.button.clicked.connect(self.run)

        #! Terminal
        self.terminal = QTextEdit()
        self.terminal.setReadOnly(True)
        self.terminal.setMaximumWidth(300)
        self.terminal.setFont(QFont("Arial", 12))
        self.terminal.setStyleSheet(
            "border: 1px solid black; border-radius: 5px; padding:
5px; color:white;"
        )
        #!Charts
        self.chart = QChart()
        self.chart.setTheme(QChart.ChartThemeDark)
        self.chartView = QChartView(self.chart)
        self.chartView.setMinimumHeight(300)
        self.chartView.setRenderHint(QPainter.Antialiasing)

        #! progress bar
        self.progress = QProgressBar(self)
        self.progress.setMinimumSize(200, 30)
        self.progress.setRange(0, 0)
        self.progress.setRange(0, 100)
        self.progress.setValue(0)
        self.progress.hide()
        self.progress.setStyleSheet("""
                QProgressBar {
                    border: 2px solid #555;
                    border-radius: 8px;
                    text-align: center;
                    background: #2b2b2b;
                    color: white;
                }
                QProgressBar::chunk {
                    background-color: #00bcd4;
                    width: 20px;
                    margin: 1px;
                }
                """)

        #! layout
        vbox = QVBoxLayout()
        vbox.setContentsMargins(50, 50, 50, 50)
        vbox.setAlignment(Qt.AlignmentFlag.AlignTop)
        vbox.setSpacing(6)
        vbox.addWidget(label)
        vbox.addWidget(self.mode_label)
        vbox.addWidget(self.select_mode)

        paramenter_box = QHBoxLayout()
        paramenter_box.setAlignment(Qt.AlignmentFlag.AlignHCenter)
        paramenter_box.setContentsMargins(0, 0, 0, 20)
        paramenter_box.setSpacing(50)
        paramenter_box.addLayout(min_box)
        paramenter_box.addLayout(max_box)
        paramenter_box.addLayout(self_box)

        vbox.addWidget(parameter_info)
        vbox.addLayout(paramenter_box)

        vbox.addWidget(l2)
        vbox.addWidget(self.select)
        vbox.addWidget(self.button)

        hbox2 = QHBoxLayout()
        hbox2.addWidget(self.terminal)
        hbox2.addWidget(self.chartView)
        vbox.addLayout(hbox2)
        centre.setLayout(vbox)

    #! drawing helper functions
    def get_range_input(self, min=10, max=100, value=0,
font=DEFAULT_FONT):
        r = QSpinBox()
        r.setFont(font)
        r.setMinimum(min)
        r.setMaximum(max)
        r.setValue(value)
        return r

    def get_label(self, text, font=DEFAULT_FONT,
align=Qt.AlignCenter):
        l = QLabel(text, self)
        l.setFont(font)
        l.setAlignment(align)
        return l

    def get_box(self, *x):
        box = QHBoxLayout()
        box.setDirection(QBoxLayout.RightToLeft)
        box.setAlignment(Qt.AlignmentFlag.AlignRight)
        box.setSpacing(6)
        for i in x:
            box.addWidget(i)
        return box


#! helper functions


    def get_selected(self):
        selected = []
        for i in range(self.select.count()):
            item = self.select.item(i)
            if item.checkState() == Qt.Checked:
                selected.append(item.text())
        return selected


##! process functions

    def run(self):
        print("running")
        self.button.setDisabled(True)

        # if SINGLE_MODE then simply run engine once and show the
result
        if (self.mode == SINGLE_MODE):
            self.run_engine(self.min.value(), self.max.value(),
                            self._size.value(), self.get_selected())

        # if MULTIPLE_MODE then run engine for each size
        if (self.mode == MULTIPLE_MODE):
            self.sizes = [int(s.strip())
                          for s in
self._size.toPlainText().split(",") if s != ""]
            self.progress.setValue(0)
            self.progress.show()
            i = 0
            for size in self.sizes:
                i += 1
                self.run_engine_Mutiple(self.min.value(),
self.max.value(),
                                        size, self.get_selected())

    def run_engine_Mutiple(self, min, max, size, algo_list):
        process = QProcess()
        process.readyReadStandardOutput.connect(
                lambda: self.process_output(process))
            process.finished.connect(lambda:
self.process_finished(size))
            process.start(ENGINE_PATH, [str(min), str(max), str(size),
*algo_list])

    def run_engine(self, min, max, size, algo_list):
        self.process = QProcess()
        self.process.readyReadStandardOutput.connect(
            lambda: self.process_output(self.process))
        self.process.finished.connect(self.process_finished)
        self.process.start(ENGINE_PATH, [str(min), str(max), str(
            size), *algo_list])
        self.progress.setValue(0)
        self.progress.show()

    def process_output(self, process):
        output = process.readAllStandardOutput().data().decode("utf-
8")
        if (output.strip().startswith("PROGRESS") and self.mode ==
SINGLE_MODE):
            self.progress.setValue(
                int((output.strip().split(" :
")[1]).split("\n")[0]))
            print(output.strip("PROGRESS : 100\n"))
        self.terminal.setText(output.strip("PROGRESS : 100\n"))

    def process_finished(self, size=0):
        if (self.mode == SINGLE_MODE):
            self.button.setDisabled(False)
            result = self.parse_results(self.terminal.toPlainText())
            sorted_result = dict(sorted(result.items(), key=lambda
x: x[1]))
            self.terminal.setText("")
            for r in sorted_result:
                self.terminal.setText(self.terminal.toPlainText(
                ) + r + " : " + str(sorted_result[r]) + "ms\n")
            self.set_chart_single(sorted_result)
            self.progress.hide()

        if (self.mode == MULTIPLE_MODE):
            result = self.parse_results(self.terminal.toPlainText())
            for r in result:
                self.result.append(
                    {"name": r, "size": size, "time": result[r]})
            self.count += 1
            self.progress.setValue((self.count /
len(self.sizes))*100)
```

```
            if (self.count == len(self.sizes)):
                self.terminal.setText("")
                total_time = {}
                for r in self.result:
                    if r["name"] not in total_time:
                        total_time[r["name"]] = 0
                    total_time[r["name"]] += r["time"]
                total_time = dict(sorted(total_time.items(),
                                         key=lambda x: x[1]))

                for t in total_time:
                    self.terminal.setText(self.terminal.toPlainText(
                    ) + t + " : " + str(total_time[t]) + "ms\n")

                self.progress.hide()
                self.button.setDisabled(False)
                self.set_chart_mutiple(self.result)
                self.result = []
                self.count = 0

    def parse_results(self, output):
        results = {}
        for line in output.split("\n"):
            if (":" not in line):
                continue
            algo, time = line.split(" : ")
            results[algo] = int(time.split("ms")[0])
        if ("PROGRESS" in results):
            results.pop("PROGRESS")
        return results

    def set_chart_mutiple(self, results):
        self.chart.removeAllSeries()
        for axis in self.chart.axes():
            self.chart.removeAxis(axis)

        # Group results by algorithm name
        grouped = defaultdict(list)
        for entry in results:
            grouped[entry["name"]].append(entry)

        # Add one QLineSeries per algorithm
        all_x, all_y = [], []
        for name, values in grouped.items():
            # sort by input size
            values = sorted(values, key=lambda x: x["size"])
            series = QLineSeries()
            series.setName(name)
            for v in values:
                series.append(v["size"], v["time"])
                all_x.append(v["size"])
                all_y.append(v["time"])
            self.chart.addSeries(series)

        self.chart.setTitle("Sorting Algorithm Runtimes (ms)")
        self.chart.setAnimationOptions(QChart.SeriesAnimations)

        # X Axis (Input Size)
        axisX = QValueAxis()
        axisX.setTitleText("Input Size")
        axisX.setLabelFormat("%d")
        axisX.setRange(min(all_x), max(all_x))
        self.chart.addAxis(axisX, Qt.AlignBottom)

        # Y Axis (Runtime)
        axisY = QValueAxis()
        axisY.setTitleText("Runtime (ms)")
        axisY.setLabelFormat("%d")
        axisY.setRange(0, max(all_y))
        self.chart.addAxis(axisY, Qt.AlignLeft)

        # Attach all series to both axes
        for series in self.chart.series():
            series.attachAxis(axisX)
            series.attachAxis(axisY)

    def set_chart_single(self, results):
        self.chart.removeAllSeries()
        for axis in self.chart.axes():
            self.chart.removeAxis(axis)

        series = QBarSeries()

        bars = []
        for algo, value in results.items():
            bar_set = QBarSet(algo)
            bar_set.append(value)
            bars.append(bar_set)
        series.append(bars)
        self.chart.addSeries(series)
        self.chart.setTitle("Sorting Algorithm Runtime (ms)")
        self.chart.setAnimationOptions(QChart.SeriesAnimations)

        # X Axis (categories = algorithm names)
        axisX = QBarCategoryAxis()
        axisX.append(["Algorithms"])
        self.chart.addAxis(axisX, Qt.AlignBottom)
        series.attachAxis(axisX)

        # Y Axis
        axisY = QValueAxis()
        axisY.setTitleText("Runtime (ms)")
        self.chart.addAxis(axisY, Qt.AlignLeft)
        series.attachAxis(axisY)


def main():
    app = QApplication(sys.argv)
    window = MainWindow()
    window.showMaximized()
    sys.exit(app.exec())


if __name__ == "__main__":
    main()
```

---------------------------main.py[end]--------------------------

-----------------------------const.py----------------------------

```
SINGLE_MODE = "Single"
MULTIPLE_MODE = "Multiple"

ENGINE_PATH = "./Engine_cpp/engine.exe"
ALGORITHMS = ["BubbleSort", "InsertionSort",
              "MergeSort", "QuickSort", "SelectionSort",
"OptimizedBubbleSort"]

# MODES
MODE_NAME = ["Comparison of Algorithms on one dataset ",
             "Comparison of Algorithms on multiple datasets"]
MODE_VALUE = {MODE_NAME[0]: SINGLE_MODE, MODE_NAME[1]: MULTIPLE_MODE}


# LIMITS

MIN_LIMIT = -10000
MAX_LIMIT = 10000
SIZE_LIMIT = 1000000
DEFAULT_SIZES = "10000, 20000, 30000, 40000, 50000"
```
-------------------------const.py[end]-------------------------

```
-------------------------engine.cpp-------------------------

#include <iostream>
#include <sstream>
#include <map>
#include <string>
#include <chrono>
#include "file_manager.h"
#include "sorting_algorithm.h"
using namespace std;

class Engine
{
private:
    File f;
    SortingAlgorithm *algo;

public:
    static const char delimiter = '\n';
    Engine(File file, SortingAlgorithm *algo)
    {
        this->f = file;
        this->algo = algo;
    }

    int run()
    {
        string content = f.read();
        vector<int> arr = string_to_vector(content);
        auto start = chrono::high_resolution_clock::now();
        algo->sort(arr);
        auto end = chrono::high_resolution_clock::now();
        content = vector_to_string(arr);
        f.write(content);
        int execution_time =
chrono::duration_cast<chrono::milliseconds>(end - start).count();
        return execution_time;
    }
    static string map_to_string(map<string, int> map)
    {
        string result = "";
        for (auto it = map.begin(); it != map.end(); it++)
        {
            result += it->first + " : " + to_string(it->second) +
"ms" + "\n";
        }
        return result;
    }
    vector<int> &string_to_vector(string str)
    {
        static vector<int> result;
        result.clear();
        string token;
        istringstream tokenStream(str);
        while (getline(tokenStream, token, delimiter))
        {
            result.push_back(stoi(token));
        }
        return result;
    }
    static string vector_to_string(vector<int> arr)
    {
        string result = "";
        for (int i = 0; i < arr.size(); i++)
        {
            result += to_string(arr[i]) + delimiter;
        }
        return result;
    }
};
class RandomNumberGenerator
{
public:
    static int generate(int min, int max)
    {
        return rand() % (max - min + 1) + min;
    }
};
class FileFiller
{
public:
    static void fill_with_random(int min, int max, int size,
vector<File> files)
    {
        string content;
        for (int i = 0; i < size; i++)
        {
            content +=
to_string(RandomNumberGenerator::generate(min, max)) +
Engine::delimiter;
        }
        for (File file : files)
        {
            file.write(content);
        }
    }
};

class Tester
{
public:
    static map<string, int> &test(vector<SortingAlgorithm *>
algorithms, int min = 0, int max = 10, int size = 1000)
    {
        vector<File> files;
        static map<string, int> results;
        results.clear();

        for (SortingAlgorithm *algo : algorithms)
        {
            File f("test_" + (*algo).name + ".txt");
            files.push_back(f);
        }
        FileFiller::fill_with_random(min, max, size, files);
        for (int i = 0; i < algorithms.size(); i++)
        {
            Engine engine(files[i], algorithms[i]);
            int execution_time = engine.run();
            cout << "PROGRESS : " + to_string((int)((i + 1.0) /
algorithms.size() * 100)) << endl;
            results.insert({algorithms[i]->name, execution_time});
        }
        return results;
    }
};
int main(int argc, char *argv[])
{
    if (argc <= 1)
    {
        cout << "No Arguments provided : Need min max size
...algorithm_list" << endl;
        return 1;
    }
    int min, max, size;
    try
    {
        min = atoi(argv[1]);
        max = atoi(argv[2]);
        size = atoi(argv[3]);
    }
    catch (exception e)
    {
        cout << "Invalid Arguments : Need min max size
...algorithm_list" << endl;
        return 1;
    }

    vector<SortingAlgorithm *> algorithms;
    for (int i = 4; i < argc; i++)
    {
        auto temp = getAlgorithm(argv[i]);
        if (temp != nullptr)
        {
            algorithms.push_back(temp);
        }
    }
    if (algorithms.size() == 0)
    {
        algorithms.push_back(new MergeSort());
        algorithms.push_back(new InsertionSort());
        algorithms.push_back(new SelectionSort());
        algorithms.push_back(new BubbleSort());
        algorithms.push_back(new QuickSort());
        algorithms.push_back(new OptimizedBubbleSort());
    }
    auto result = Tester::test(algorithms, min, max, size);
    cout << Engine::map_to_string(result) << endl;
    auto resultFile = File("results.txt");

    resultFile.append("\n___________________________________\n");
    resultFile.append("\n" + to_string(min) + "  " + to_string(max) +
"  " + to_string(size) + "\n");
    resultFile.append(Engine::map_to_string(result));
    resultFile.append("\n___________________________________\n");

    return 0;
}
```

-----------------------engine.cpp[end]-----------------------

--------------------sorting_algorithm.h ------------------

```
#ifndef SORTING_ALGORITHM
#define SORTING_ALGORITHM
#include <vector>
using namespace std;

class SortingAlgorithm
{
```

```cpp
public:
    string name;
    virtual void sort(vector<int> &arr) = 0;
    virtual ~SortingAlgorithm() {}
};

class BubbleSort : public SortingAlgorithm
{
public:
    BubbleSort()
    {
        this->name = "BubbleSort";
    }
    void sort(vector<int> &arr) override
    {
        // cout << "BubbleSort" << endl;
        for (int i = 0; i < arr.size() - 1; i++)
        {
            for (int j = 0; j < arr.size() - i - 1; j++)
            {
                if (arr[j] > arr[j + 1])
                {
                    swap(arr[j], arr[j + 1]);
                }
            }
        }
    };
};
class OptimizedBubbleSort : public SortingAlgorithm
{
public:
    OptimizedBubbleSort()
    {
        this->name = "OptimizedBubbleSort";
    }
    void sort(vector<int> &arr) override
    {
        // cout << "OptimizedBubbleSort" << endl;
        bool swapped = true;

        for (int i = 0; i < arr.size() - 1 && swapped; i++)
        {
            swapped = false;
            for (int j = 0; j < arr.size() - i - 1; j++)
            {
                if (arr[j] > arr[j + 1])
                {
                    swap(arr[j], arr[j + 1]);
                    swapped = true;
                }
            }
        }
    };
};

class InsertionSort : public SortingAlgorithm
{
public:
    InsertionSort()
    {
        this->name = "InsertionSort";
    }
    void sort(vector<int> &arr) override
    {
        // cout << "InsertionSort" << endl;
        for (int i = 1; i < arr.size(); 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;
        }
    };
};

class SelectionSort : public SortingAlgorithm
{
public:
    SelectionSort()
    {
        this->name = "SelectionSort";
    }
    void sort(vector<int> &arr) override
    {
        // cout << "SelectionSort" << endl;
        for (int i = 0; i < arr.size() - 1; i++)
        {
            int min = i;
            for (int j = i + 1; j < arr.size(); j++)
            {
                if (arr[j] < arr[min])
                {
                    min = j;
                }
            }
            swap(arr[i], arr[min]);
        }
    }
};

class MergeSort : public SortingAlgorithm
{
public:
    MergeSort()
    {
        this->name = "MergeSort";
    }
    void sort(vector<int> &arr) override
    {
        static bool first = true;
        if (first)
        {
            // cout << "MergeSort" << endl;
            first = false;
        }
        if (arr.size() <= 1)
            return;
        int mid = arr.size() / 2;
        vector<int> left(arr.begin(), arr.begin() + mid);
        vector<int> right(arr.begin() + mid, arr.end());
        sort(left);
        sort(right);
        merge(arr, left, right);
    };
    void merge(vector<int> &arr, vector<int> &left, vector<int>
&right)
    {
        int i = 0, j = 0, k = 0;
        while (i < left.size() && j < right.size())
        {
            if (left[i] < right[j])
            {
                arr[k++] = left[i++];
            }
            else
            {
                arr[k++] = right[j++];
            }
        }
        while (i < left.size())
        {
            arr[k++] = left[i++];
        }
        while (j < right.size())
        {
            arr[k++] = right[j++];
        }
    }
};

class QuickSort : public SortingAlgorithm
{
public:
    QuickSort()
    {
        this->name = "QuickSort";
    }
    void sort(vector<int> &arr) override
    {
        // cout << "QuickSort" << endl;
        quickSort(arr, 0, arr.size() - 1);
    };
    void quickSort(vector<int> &arr, int low, int high)
    {
        if (low < high)
        {
            int pi = partition(arr, low, high);
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }
    int partition(vector<int> &arr, int low, int high)
    {
        int pivot = arr[high];
        int i = low - 1;
        for (int j = low; j <= high - 1; j++)
        {
            if (arr[j] < pivot)
            {
                i++;
                swap(arr[i], arr[j]);
            }
        }
        swap(arr[i + 1], arr[high]);
        return i + 1;
    }
};
```

```
SortingAlgorithm *getAlgorithm(string name)
{
    if (name == "MergeSort")
        return new MergeSort();
    if (name == "InsertionSort")
        return new InsertionSort();
    if (name == "SelectionSort")
        return new SelectionSort();
    if (name == "BubbleSort")
        return new BubbleSort();
    if (name == "QuickSort")
        return new QuickSort();
    if (name == "OptimizedBubbleSort")
        return new OptimizedBubbleSort();
    return nullptr;
}

#endif
```

--------------- sorting_algorithm.h[end]---------------

----------------------- file_manager.h---------------------

```
#ifndef FILE_MANAGER
#define FILE_MANAGER
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
using namespace std;
class File
{
private:
    string path;

public:
    File() {};
    File(string path)
    {
        this->path = path;
        FILE *file = fopen(path.c_str(), "a");
        if (file == NULL)
        {
            cout << "Error opening file" << endl;
        }
        fclose(file);
    }
    string read()
    {
        ifstream file_stream(path);
        if (!file_stream)
        {
            cout << "Error reading file" << endl;
        }
        string content, line;

        while (getline(file_stream, line))
        {
            content += line + "\n";
        }
        file_stream.close();
        return content;
    };
    bool write(string content)
    {
        ofstream file_stream(path);
        if (!file_stream)
        {
            cout << "Error finding file" << endl;
            return false;
        }
        file_stream << content;
        file_stream.close();
        return true;
    };
    bool append(string content)
    {
        ofstream file_stream(path, ios::app);
        if (!file_stream)
        {
            cout << "Error finding file" << endl;
            return false;
        }
        file_stream << content;
        file_stream.close();
        return true;
    }
};

#endif
```

------------------ file_manager.h[end]-----------------

6.Results and Analysis

Objective

The purpose of this experiment is to evaluate and compare the performance of different sorting algorithms implemented in C++ through the developed *Sorting Benchmarking Tool*. The analysis focuses on execution time with varying input sizes to identify efficiency patterns and confirm theoretical time complexities.

6.1. Single Data Set Mode (Algorithm Comparison)

Objective

To compare the execution time of multiple sorting algorithms on the same dataset to determine relative performance and efficiency.

Experimental Setup

- **Dataset Size:** 10,000 elements
- **Value Range:** 0 – 1000
- **Algorithms Tested:**
  Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Quick Sort, Optimized Bubble Sort

Result

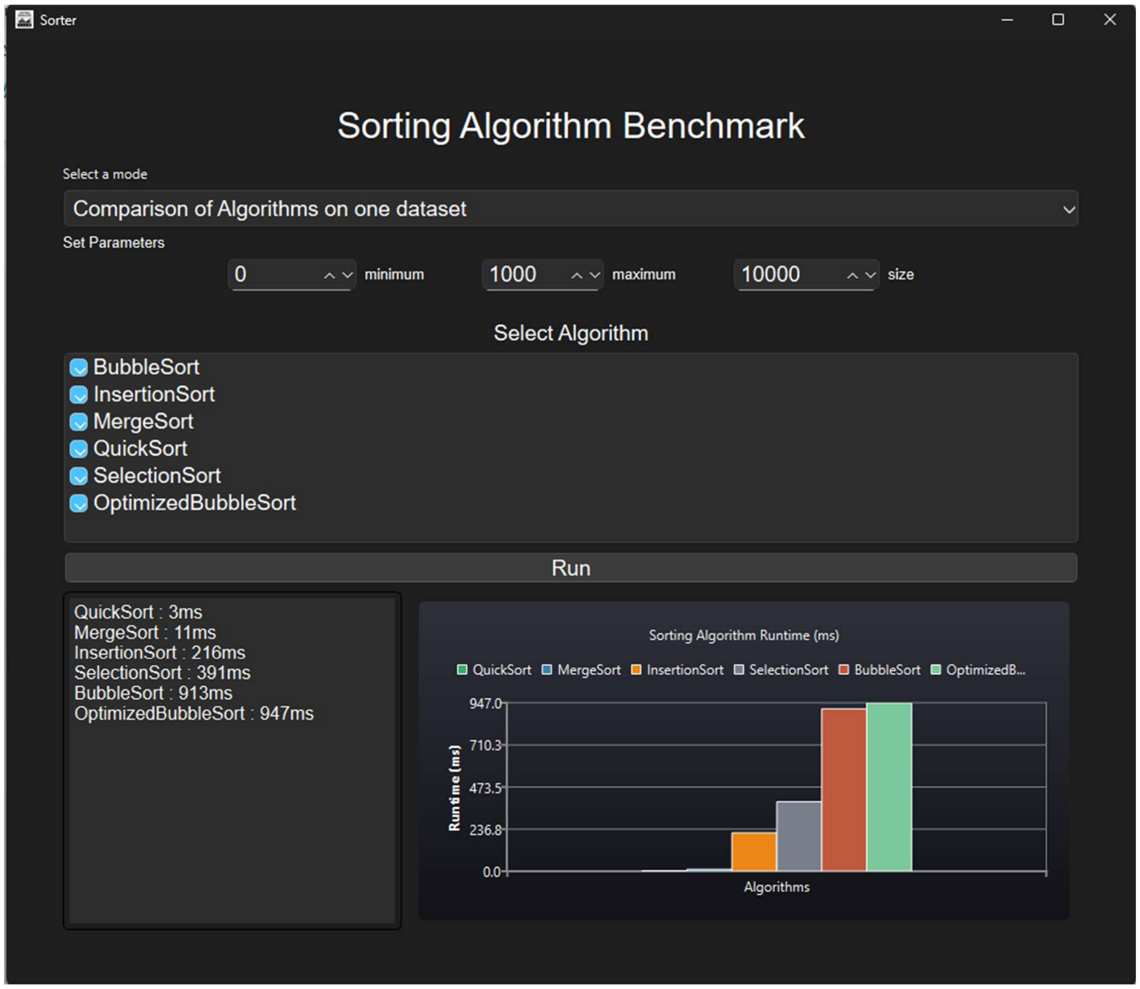
Algorithm	Runtime (ms)	Theoretical Complexity
Quick Sort	3	O(n log n)
Merge Sort	11	O(n log n)
Insertion Sort	216	$O(n^2)$
Selection Sort	391	$O(n^2)$
Bubble Sort	913	$O(n^2)$
Optimized Bubble Sort	947	$O(n^2)$

Analysis

- **Quick Sort** consistently performed the best due to its efficient divide-and-conquer strategy.
- **Merge Sort** performed similarly well, confirming its predictable O(n log n) behaviour.

- **Quadratic algorithms** (Bubble, Insertion, Selection) showed significantly higher runtimes.
- The **optimized version of Bubble Sort** performed slightly worse than the normal because of the overhead of the flag.

GUI



```
QuickSort : 3ms
MergeSort : 11ms
InsertionSort : 216ms
SelectionSort : 391ms
BubbleSort : 913ms
OptimizedBubbleSort : 947ms
```

6.2. Multiple Data Set Mode (Scalability Test)

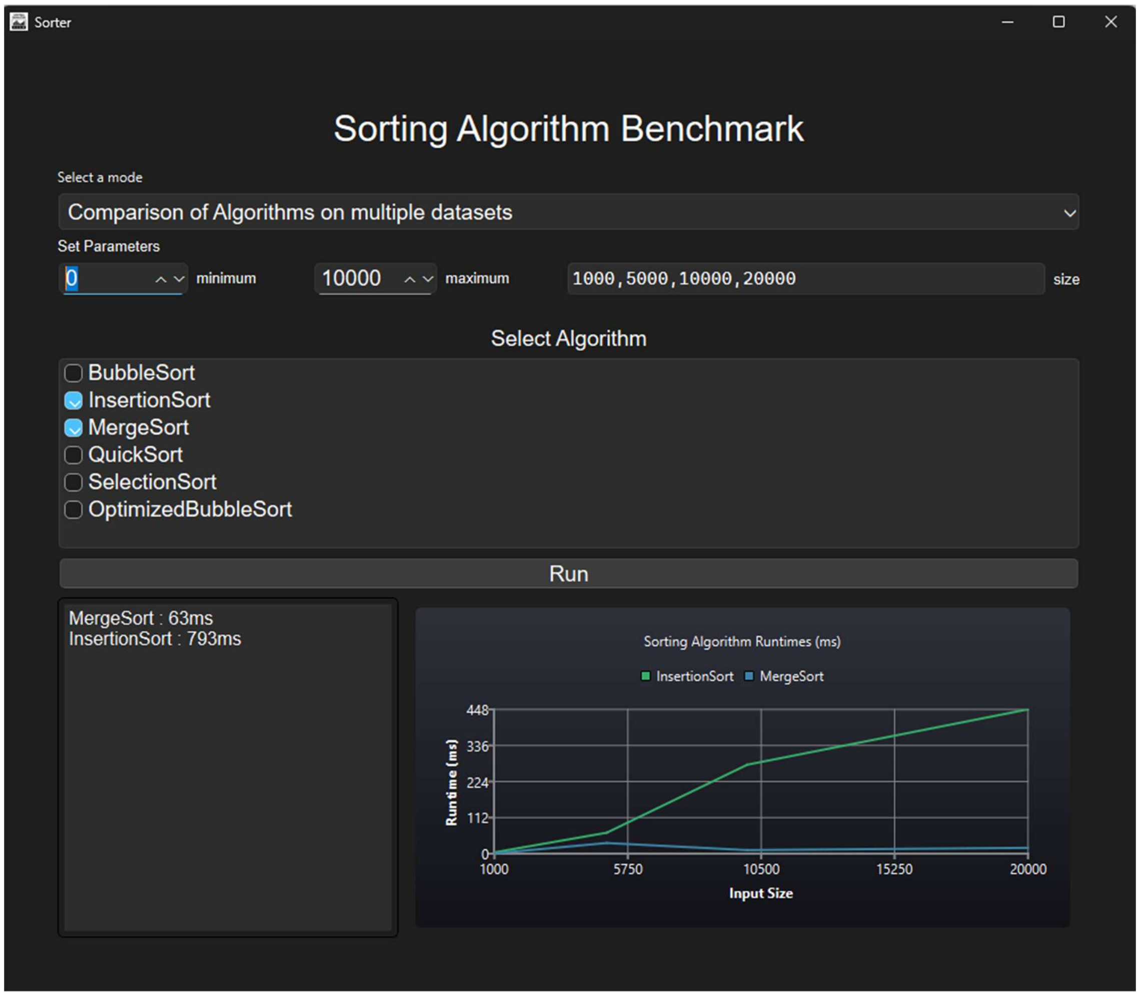
Objective

To evaluate the scalability of the implemented sorting algorithms, tests were conducted on datasets of increasing sizes. The performance was measured in milliseconds (ms) for Merge Sort and Insertion Sort, representing their efficiency as input size grows.

Experimental Setup

- **Algorithm Selected:** Merge and Insertion Sort
- **Dataset Sizes:** 1,000, 5,000, 10,000, 20,000 elements
- **Value Range:** 0 – 10,000

GUI



Sorter

Sorting Algorithm Benchmark

Select a mode: Comparison of Algorithms on multiple datasets

Set Parameters: minimum 0, maximum 10000, size 1000,5000,10000,20000

Select Algorithm

- [ ] BubbleSort
- [✓] InsertionSort
- [✓] MergeSort
- [ ] QuickSort
- [ ] SelectionSort
- [ ] OptimizedBubbleSort

Run

```
MergeSort : 63ms
InsertionSort : 793ms
```

Result

Input Size (n)	Insertion Sort Runtime (ms)	Merge Sort Runtime (ms)
1,000	4	1
5,000	65	33
10,000	276	11
20,000	448	18

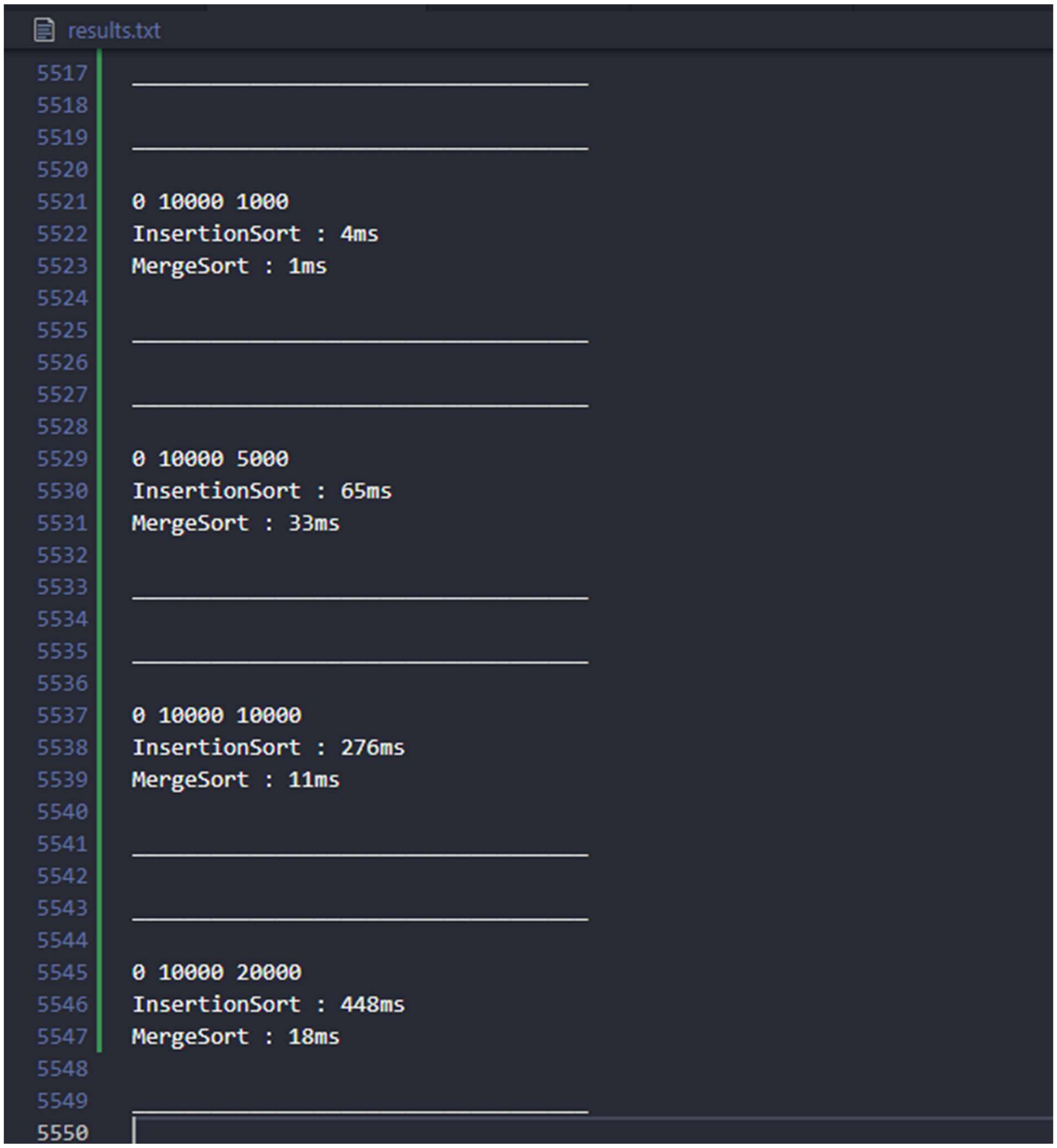
Observations

The results obtained from the benchmarking tool indicate a clear performance gap between the two algorithms as the dataset size increases.

- For smaller datasets (1,000 elements): Both algorithms perform efficiently, with minimal difference in execution time.
- For medium to large datasets (5,000–20,000 elements): Merge Sort maintains a relatively low and consistent runtime, while Insertion Sort exhibits a rapid increase in processing time.

Interpretation

- **Merge Sort** utilizes a divide-and-conquer strategy with an average time complexity of **$O(n \log n)$**, making it more scalable for larger datasets.
- **Insertion Sort** follows a sequential comparison and shifting approach with **$O(n^2)$** time complexity, which causes performance degradation as data size grows.

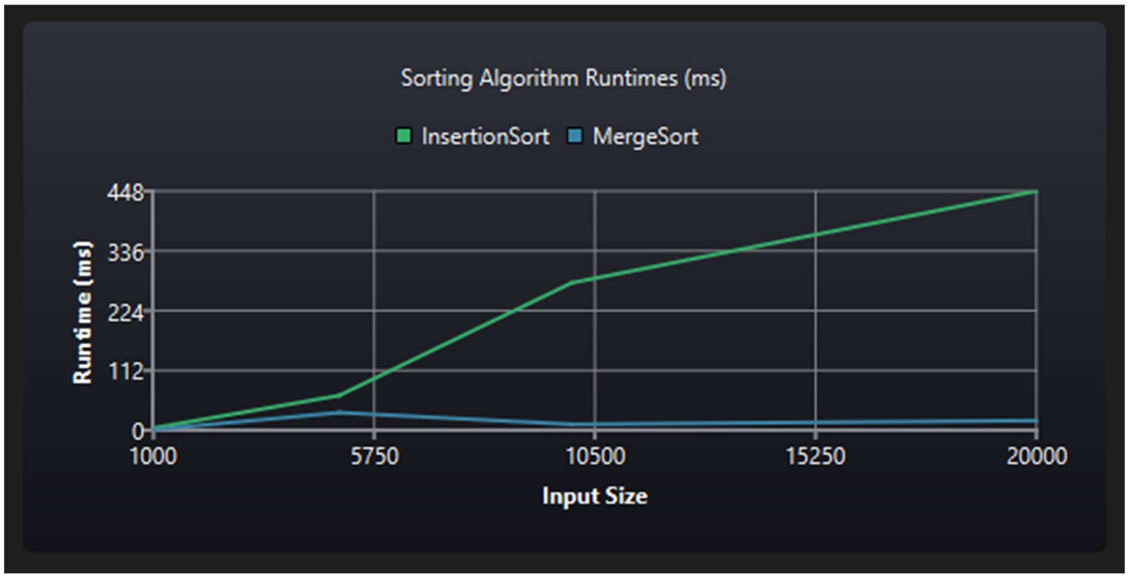


results.txt

```
___________________________________

___________________________________

0 10000 1000
InsertionSort : 4ms
MergeSort : 1ms

___________________________________

___________________________________

0 10000 5000
InsertionSort : 65ms
MergeSort : 33ms

___________________________________

___________________________________

0 10000 10000
InsertionSort : 276ms
MergeSort : 11ms

___________________________________

___________________________________

0 10000 20000
InsertionSort : 448ms
MergeSort : 18ms

___________________________________

```

Information stored in result.txt file

GRAPH



The graph above illustrates the runtime comparison between **Insertion Sort** and **Merge Sort** across varying input sizes. As seen, **Insertion Sort** shows a steep and nearly linear increase in runtime as the dataset size grows, indicating poor scalability due to its $O(n^2)$time complexity. In contrast, **Merge Sort** maintains consistently low runtimes even as input size increases, demonstrating its superior efficiency and scalability with $O(n\log n)$performance. This clearly highlights that while Insertion Sort may perform adequately on small datasets, Merge Sort is far better suited for larger datasets.

6.3. Comparison Between Merge Sort and Quick Sort (Effect of Data Distribution)

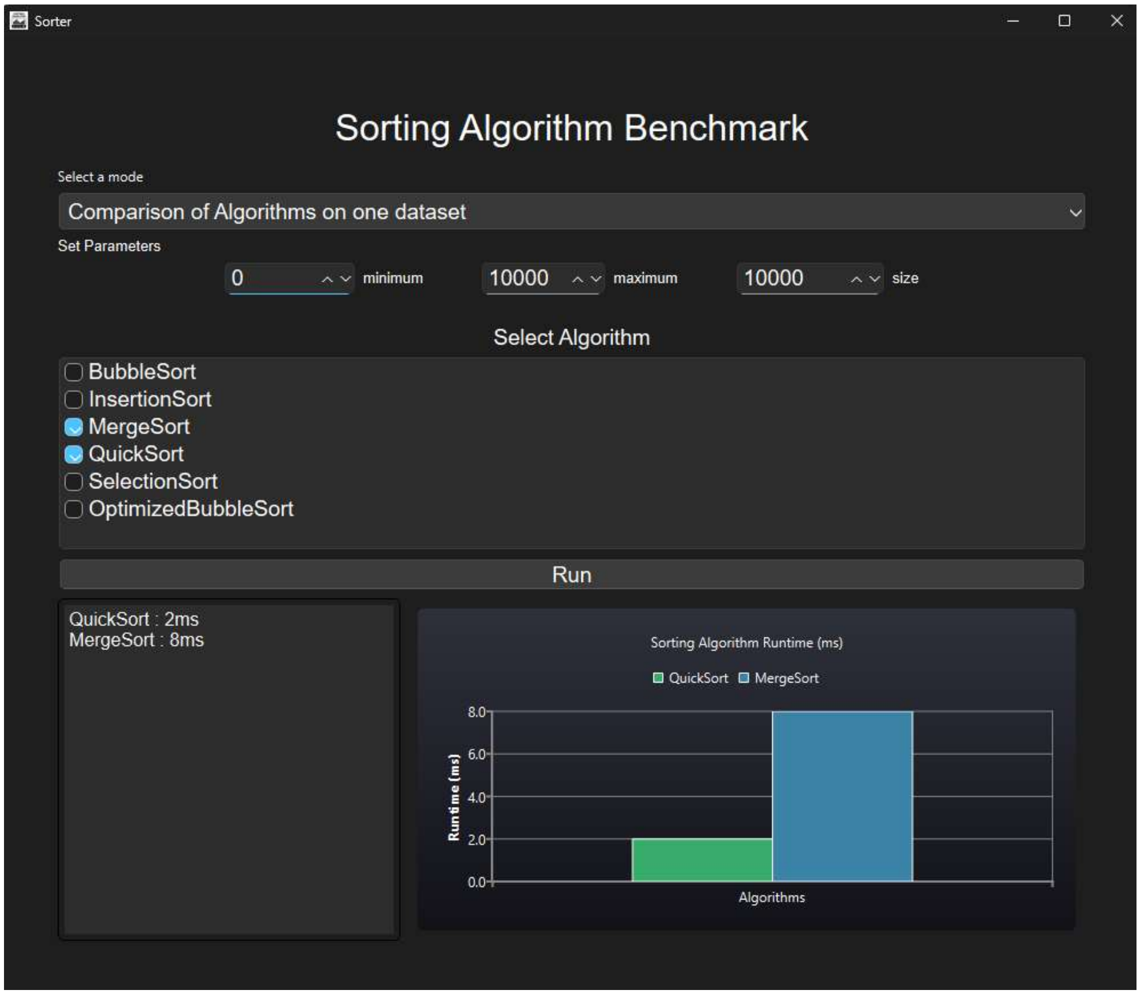
Objective:
To analyse how **data distribution** affects the performance of **Merge Sort** and **Quick Sort** algorithms.

Experimental Setup:

- **Algorithms Compared:** Merge Sort and Quick Sort
- **Dataset Size:** 10,000 elements
- **Value Range:** 0 – 10,000
- **Test Scenarios:**
    1. **Random Data** → Uniformly distributed numbers between 0 and 10,000
    2. **Nearly Sorted Data** → Values between 0 and 1 (simulating almost sorted input)

GUI

Random Data



Sorter

Sorting Algorithm Benchmark

Select a mode: Comparison of Algorithms on one dataset

Set Parameters: 0 minimum, 10000 maximum, 10000 size

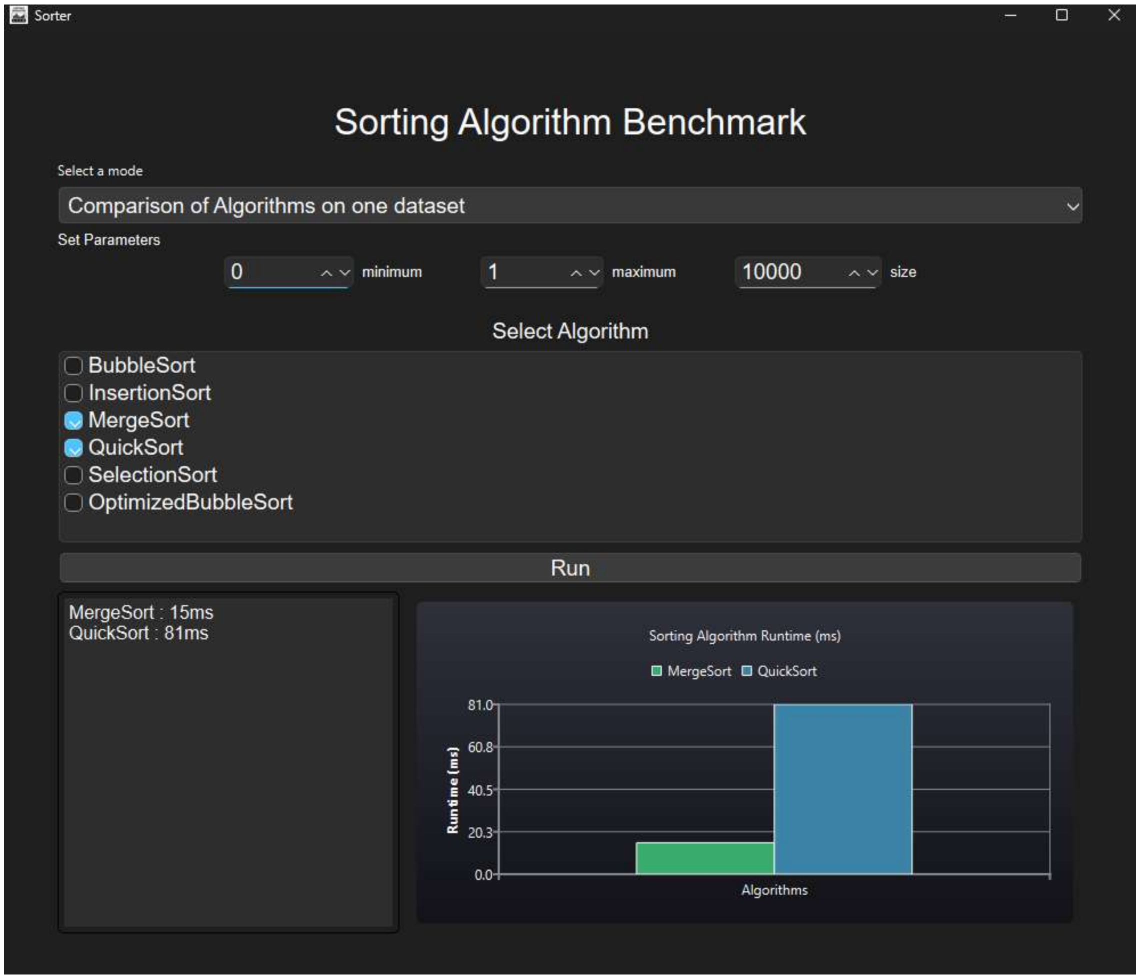
Select Algorithm

- [ ] BubbleSort
- [ ] InsertionSort
- [✓] MergeSort
- [✓] QuickSort
- [ ] SelectionSort
- [ ] OptimizedBubbleSort

Run

```
QuickSort : 2ms
MergeSort : 8ms
```

Nearly Sorted Data


Sorter

Sorting Algorithm Benchmark

Select a mode

Comparison of Algorithms on one dataset

Set Parameters

0 minimum 1 maximum 10000 size

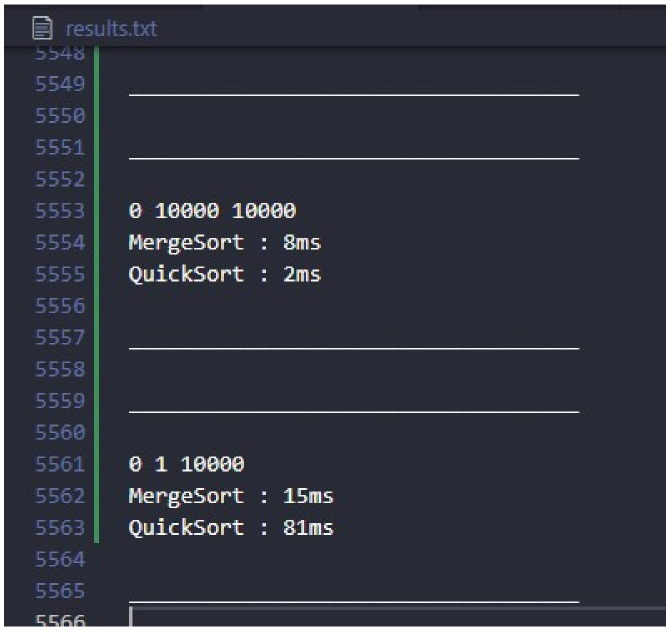
Select Algorithm

- [ ] BubbleSort
- [ ] InsertionSort
- [x] MergeSort
- [x] QuickSort
- [ ] SelectionSort
- [ ] OptimizedBubbleSort

Run

```
MergeSort : 15ms
QuickSort : 81ms
```

Exact Stats in **result.txt**


results.txt

```
______________________________________

______________________________________

0 10000 10000
MergeSort : 8ms
QuickSort : 2ms

______________________________________

______________________________________

0 1 10000
MergeSort : 15ms
QuickSort : 81ms

______________________________________
```

Result

Data Distribution	Merge Sort Runtime (ms)	Quick Sort Runtime (ms)	Observation
Random (0 – 10,000)	8	2	Quick Sort performs faster on random data due to efficient average-case partitioning.
Nearly Sorted (0 – 1)	15	81	Quick Sort runtime increases drastically, showing worst-case behaviour on nearly sorted data, while Merge Sort remains stable.

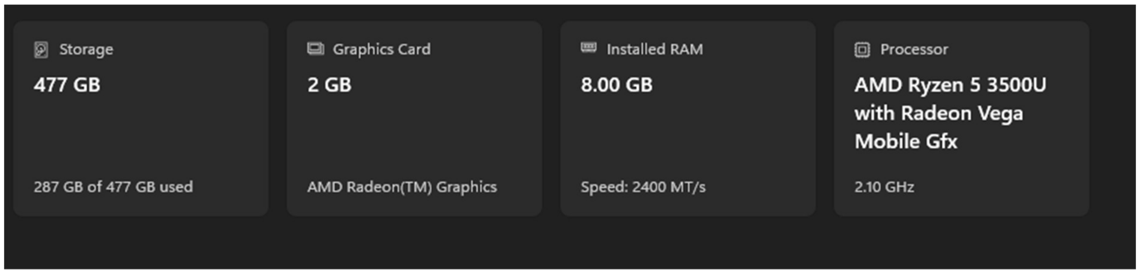
Analysis:

- When the dataset is **random**, Quick Sort outperforms Merge Sort, thanks to its in-place partitioning and smaller memory overhead.
- For **nearly sorted data**, Quick Sort's pivot selection leads to unbalanced partitions, causing recursive depth to increase and performance to degrade to near $O(n^2)$.
- Merge Sort's divide-and-conquer approach, independent of data distribution, maintains its consistent $O(n\log n)$runtime.

Conclusion:
Merge Sort offers **predictable and stable performance** regardless of input order, while Quick Sort shows **high variability** – being faster for random datasets but much slower for nearly sorted ones.
This highlights Merge Sort's robustness and Quick Sort's data-sensitivity.

-------------------System Specs Used in testing

- **Device Model:** Dell Inspiron 15 3515
- **Processor:** AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx @ 2.10 GHz
- **Graphics Card:** AMD Radeon(TM) Graphics (2 GB VRAM)
- **Installed RAM:** 8.00 GB (5.89 GB usable)
- **Storage Capacity:** 477 GB (287 GB used)
- **System Type:** 64-bit Operating System, x64-based processor
- **Operating System:** Windows 11



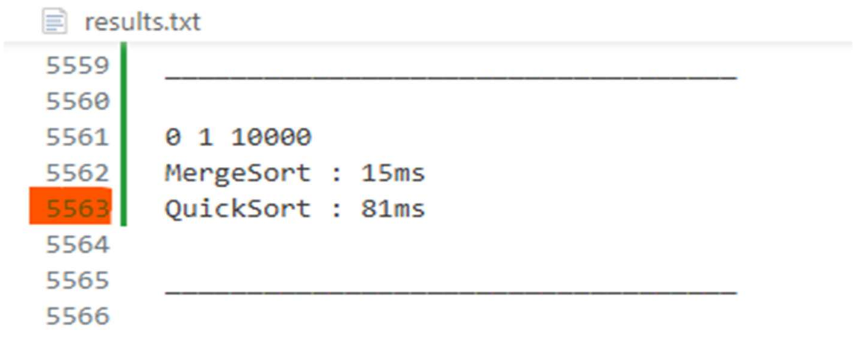
Storage	Graphics Card	Installed RAM	Processor
477 GB	2 GB	8.00 GB	AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx
287 GB of 477 GB used	AMD Radeon(TM) Graphics	Speed: 2400 MT/s	2.10 GHz

Conclusion

A total of **318 benchmark tests** were executed, each involving **6 sorting algorithms**. This amounts to

$318 \times 6 = 1908$ individual algorithm runs.

All outputs were automatically logged and saved in a file named **result.txt**, which serves as the central record for result analysis and verification.



results.txt

```
_______________________________________

0 1 10000
MergeSort : 15ms
QuickSort : 81ms

_______________________________________
```

The dataset comprised over **5,500 lines** at the end (without filtering and trimming)

This comprehensive testing allowed consistent comparisons across varying input sizes and data distributions. Results confirmed that **quadratic-time algorithms** (Bubble, Selection, and Insertion Sort) perform adequately only for small datasets, whereas **Merge Sort** and **Quick Sort** exhibit near-linear scalability with increasing data volume — closely aligning with their theoretical $O(n\log n)$complexity.

Overall, this project successfully shows how practical benchmarking helps in understanding the real performance of sorting algorithms beyond their theoretical complexity. By collecting and analysing real execution data, it bridges the gap between algorithm theory and actual system performance.