Weekly Homework Report #3

Tran Van Tan Khoi

April 7, 2025

1 Introduction

In this report, I attempt to demonstrate how sorting algorithms' running time compare when organizing integers data in ascending order. I will show how I achieved in getting the results using only C++ source files and libraries. At the same time, I attempt to give a deep dive into different categories of sorting algorithms, how each algorithm performs in certain cases, and their possible applications. Stay tuned as I show you, a race to show who is the fastest in replacing and reorder positive values in a particular order for convenient use in the future.

All of the files of this project, including this report, is available at my Github repository.

2 Overview

This experiment covers basic key aspects, including C++ implementations of sorting algorithms; a way to generate different types of data that needs to be reorganize; a way to measure the running time and all the number of comparisons; and a way to compile the results into a table of data, for the convenience of drawing charts so we can compare and judge the competitors. However, this experiment is lacking in a proper benchmark library (e.g. Google's Benchmark Library) and a generic implementation that support any comparable data type and any compare function. With that being said, the results should still give a good demonstration of how these algorithms perform in the real world.

2.1 Structure

This project includes numerous folders and files, organized as follow:

- **bin**: This folder contains the binary files, `datagen.exe`, `main.exe`, and `sorter.exe`. The purpose of each binary file is explained later with their respective C++ source file counterpart.
- **data**: This folder contains various `.txt` files and a `benchmark.csv` file that contain intermediary sorting data and all running time results.
- **docs**: This folder contains various files for a LaTeX project that compiles this very report you are reading, which beside discussing the resulting running times and number of comparisons of sorting algorithms, also explaining how this project works, how you can import your own sorting data and experiment with each algorithm implemented here.
- **libs**: This folder includes the header files for the sorting algorithms and methods of generating data for testing.
- **logs**: This folder includes the logs of running programs.
- **util**: This folder includes source files for supporting running the project.
- `main.cpp`: The main source file that will execute the majority actions of the project.

```
root
├── bin
│   ├── datagen.exe
│   ├── main.exe
│   └── sorter.exe
├── data
│   ├── benchmark.csv
│   ├── tmp
│   ├── input.txt
│   ├── answer.txt
│   └── <name>.txt
├── docs
│   ├── report.tex
│   └── report.pdf
├── libs
│   ├── include
│   ├── src
│   └── tests
├── logs
├── util
│   ├── datagen.cpp
│   └── sorter.cpp
└── main.cpp
```

Figure 1: The folder structure

2.2 Tests

This experiment is performed on Windows 11 operating system on Performance mode, build 23H2, running on Intel's Core Ultra 7 155H (4.8 GHz/24 MB), 16 GB LPDDR5X RAM, and 1TB M.2 NVMe PCIe 4.0 SSD. Please consult the table below.

Processor	Intel Core™ Ultra 7 155H (4.8 GHz Boost, 24 MB Cache)
Memory	16 GB 7467 MT/s LPDDR5X
Storage	1 TB NVMe PCIe 4.0
Operating System	Windows 11 Build 23H2, Performance Mode
Compiler	GNU G++23

2.3 Usage

sorter.cpp

Build the source file with the following command (at root folder):

```
g++ .\util\sorter.cpp .\libs\src\*.cpp -o .\bin\sorter -O2
```

And run with

```
.\bin\sorter
```

Optional flags:

- **-a**: Choosing which algorithm to run. C++ `std::sort` is run by default.
- **-i**: Direct input stream from file given by path.
- **-o**: Direct output stream to file given by path.

2.3.1 datagen.cpp

Build the soruce file with the following command (at root folder):

```
g++ .\util\datagen.cpp .\libs\tests\*.cpp -o .\bin\datagen -O2
```

And run with

```
.\bin\datagen
```

Optional flags:

- **-s**: The size of the data to be generated (at most $5 \cdot 10^6$)
- **-t**: The type of data to be generated (`random`, `sorted`, `reversed`, `nearly`, or `repeated`);
- **-o**: Direct the output stream to file given by path.

main.cpp

Build the soruce file with the following command (at root folder):

```
g++ main.cpp .\libs\src\*.cpp .\libs\tests\*.cpp  -o .\bin\main -O2
```

And run with

```
.\bin\main
```

which will run all sorting algorithms with all data types and sizes, storing each resulting run in `data/benchmark.csv`.

2.4 Storing Data

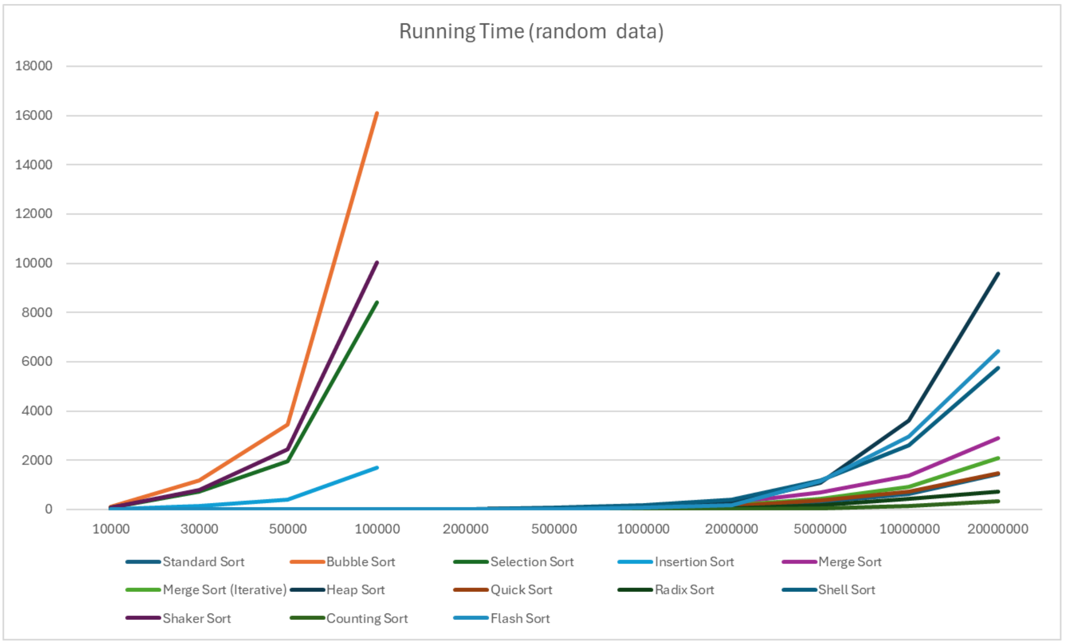
The input data and the sorted output of each algorithm is stored in the `data` folder.

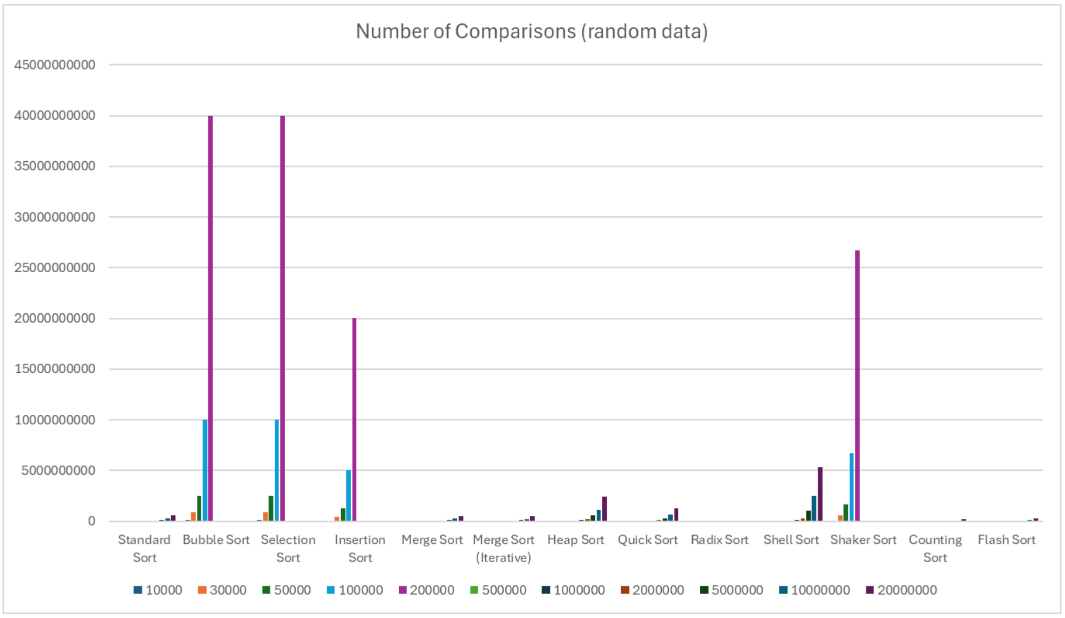
3 Algorithms

1. Bubble Sort
2. Shaker Sort
3. Selection Sort
4. Insertion Sort
5. Heap Sort
6. Merge Sort (recursive and iterative variant)
7. Quick Sort
8. Standard Sort (`std::sort` from C++ STL)
9. Radix Sort
10. Counting Sort
11. Shell Sort
12. Flash Sort

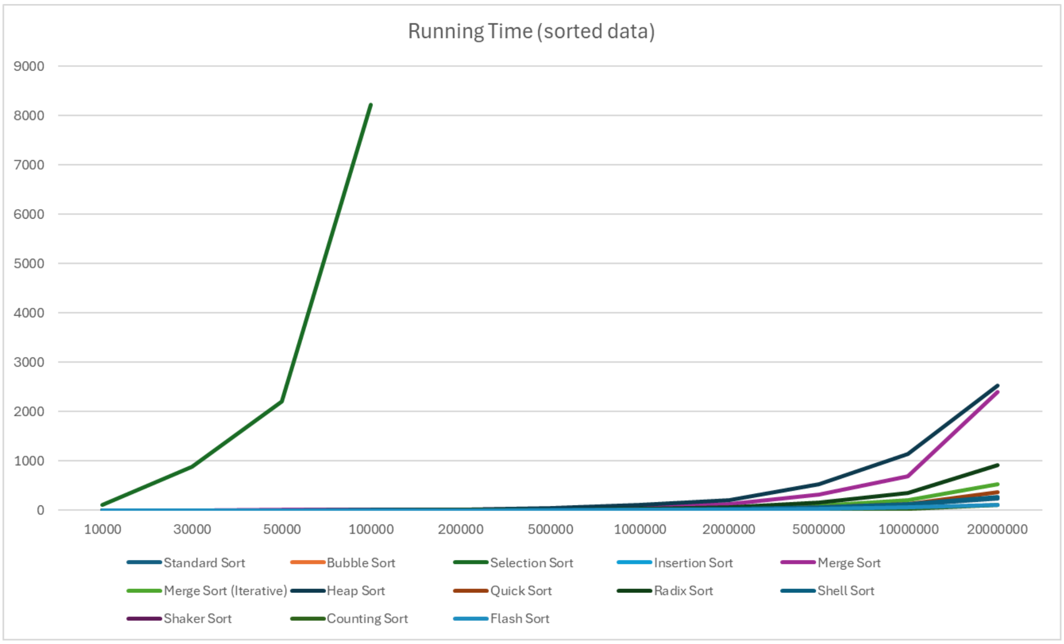
4 Results

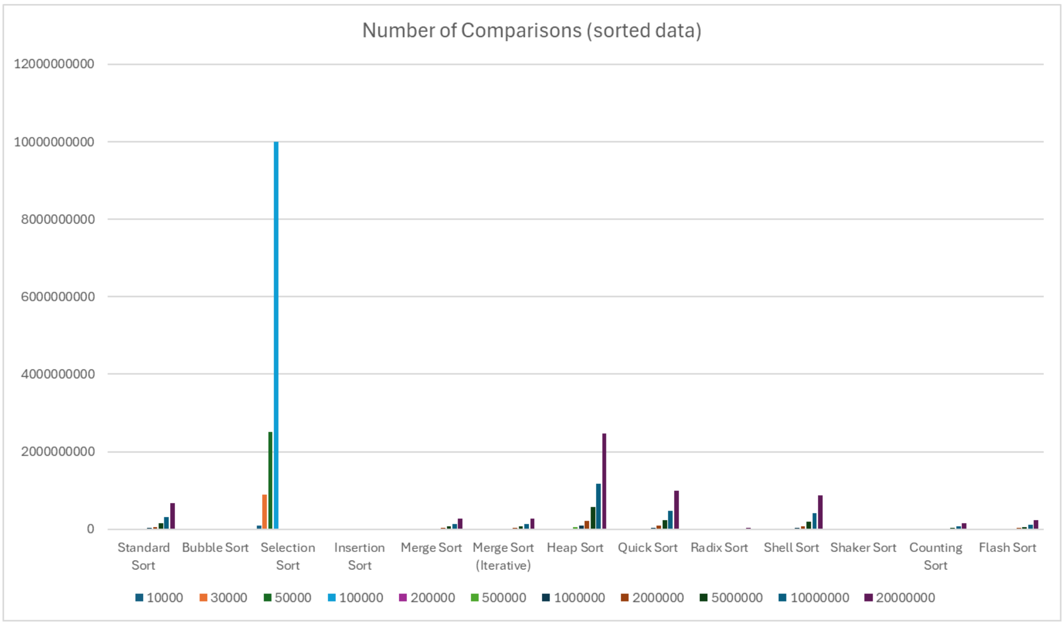
4.1 Random Data



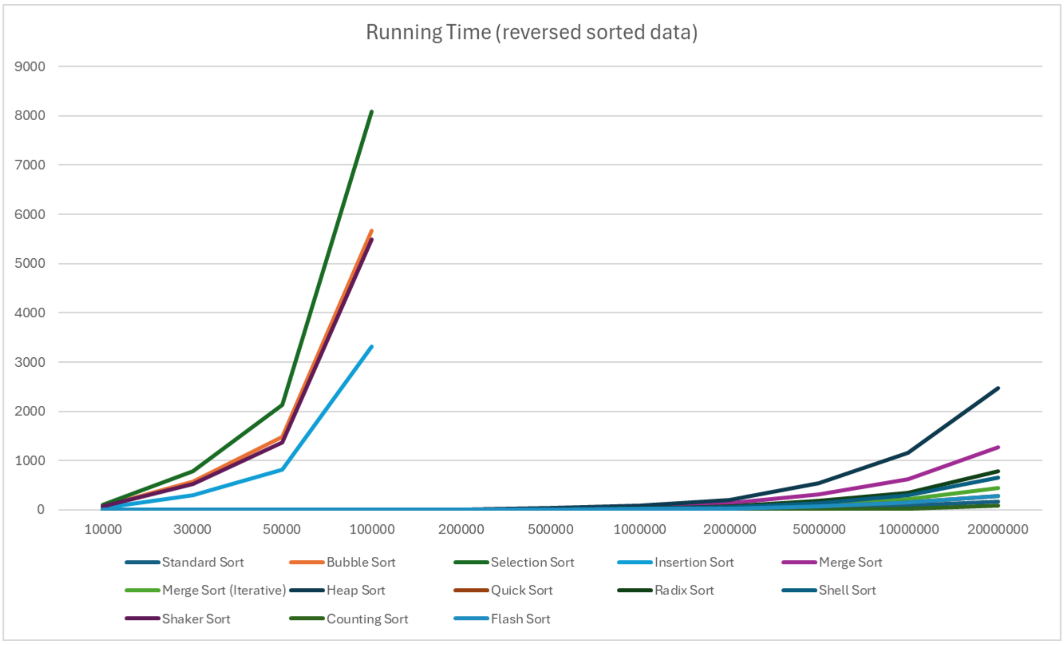


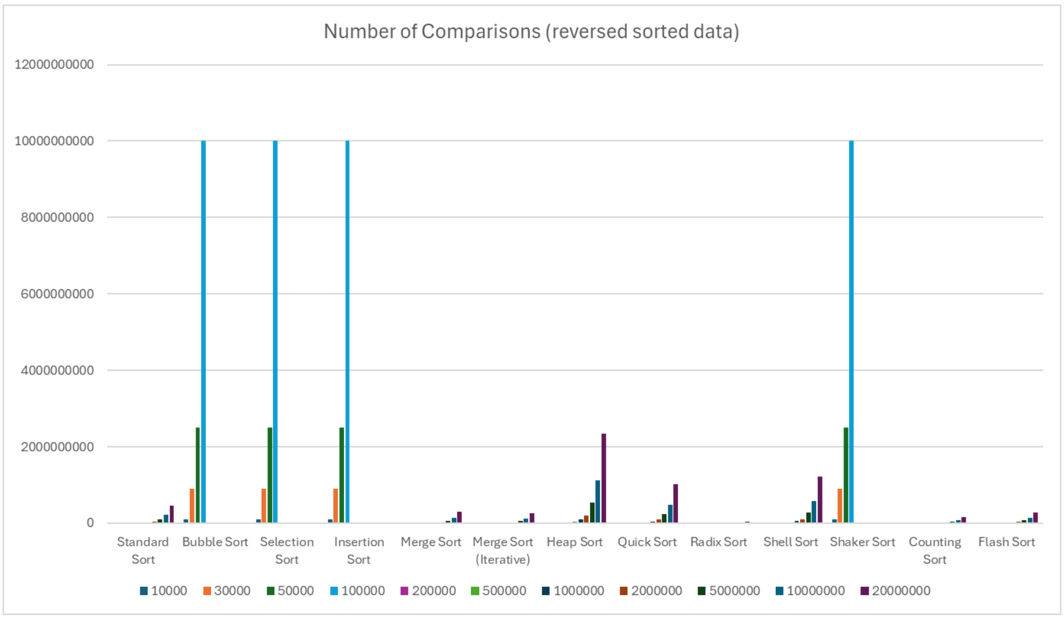
4.2 Sorted Data



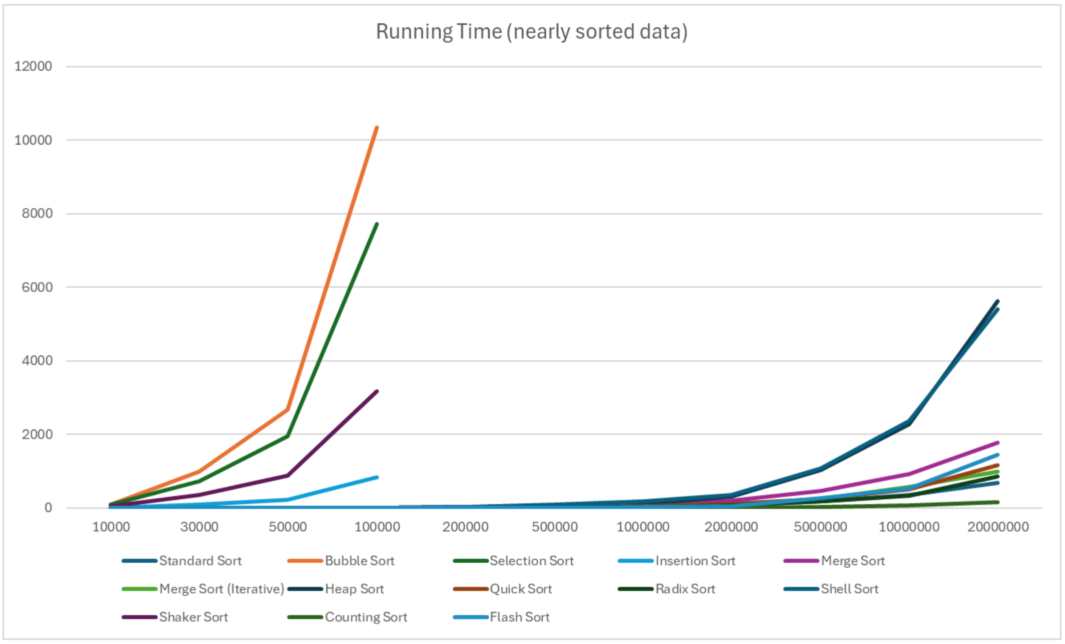


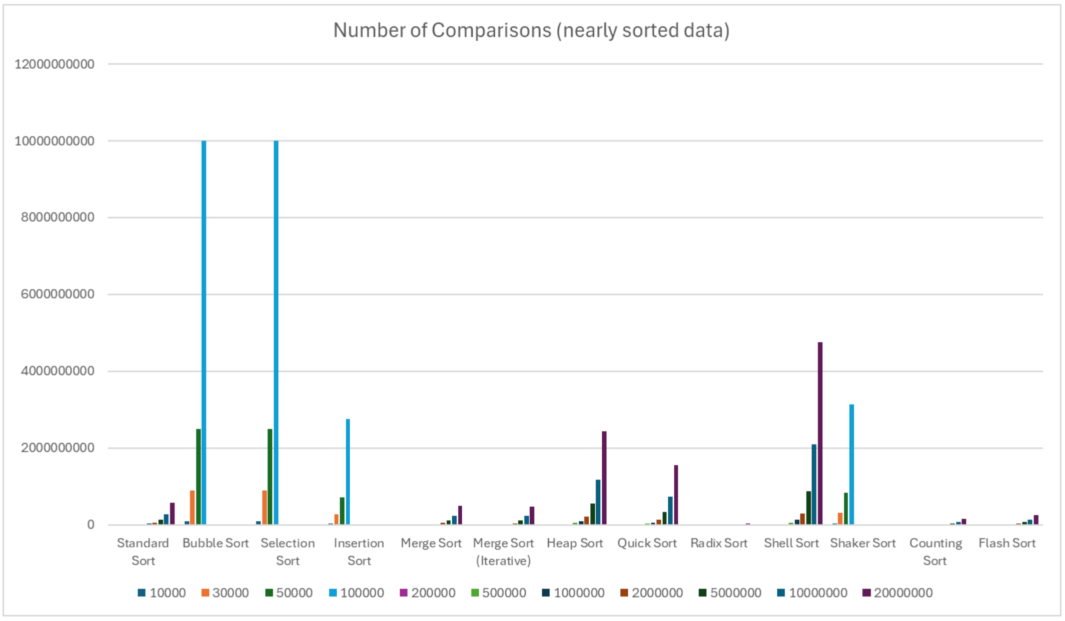
4.3 Reversed Sorted Data



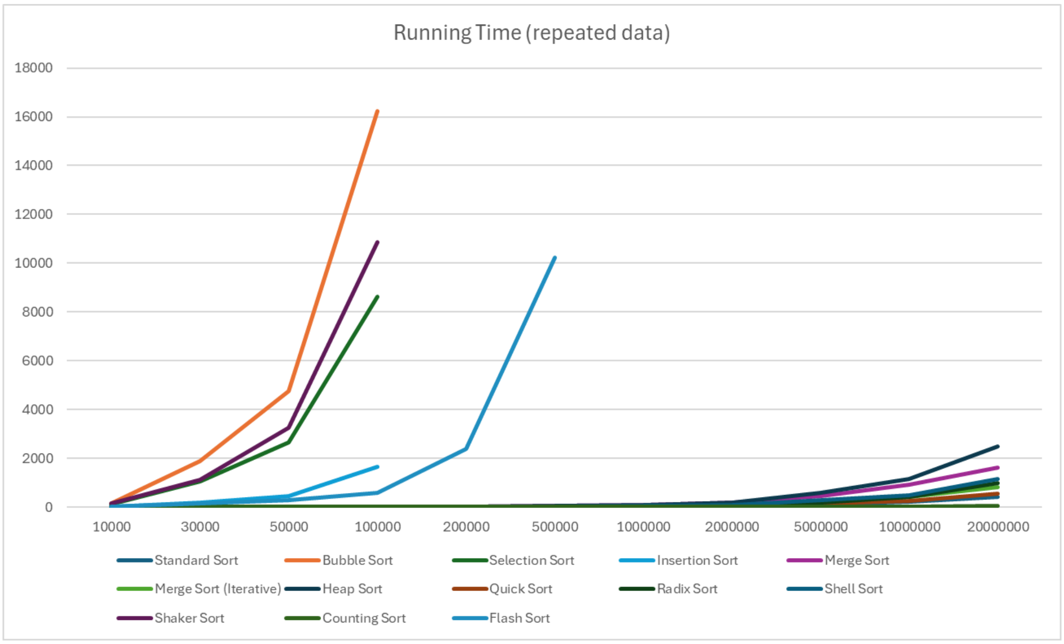


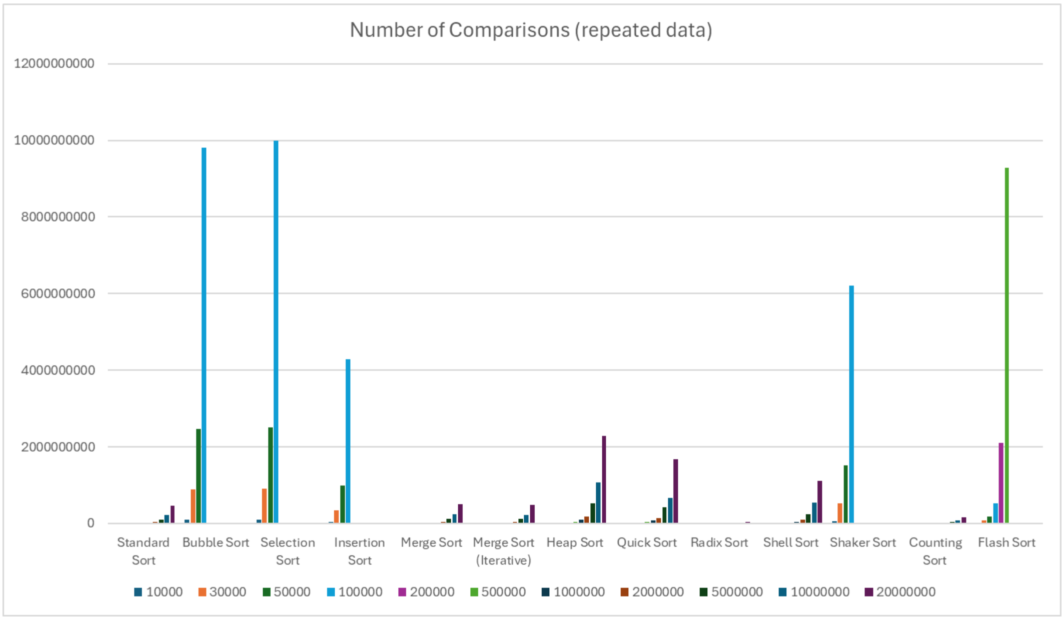
4.4 Nearly Sorted Data





4.5 Data with Many Duplicates





5 Conclusion

This experiment demonstrated how different sorting algorithms perform in many typical scenarios in terms of running time and number of comparisons made. The quadratic solutions are simple, easy to implement and run fast for small data with little to no overhead. However, they start to fall behind when the data is significantly large. This is where faster algorithms are used instead. Quasilineaer algorithms are compatible with any types of data and act as "general-purpose sorting tool" but have some drawbacks such as overhead in recursion calls or extra memory. Linear algorithms typically run the fastest, but they usually only work with certain type of data.

However, this experiment failed to demonstrate other aspect of sorting data, such as comparable data types other than `int`, and memory footprint in certain environments.