Dr. Beilei Jiang

Project 1- Sorting Algorithms

CSCE 5150: Analysis of Computer Algorithms

Group 16: Spring 2025

Amrit Adhikari AmritAdhikari@my.unt.edu 11363898
Naga Veera Subhash Alapati Nagaveerasubhashalapati@my.unt.edu 11740600
Maneesh Raja Pagadala ManeeshRajaPagadala@my.unt.edu 11815718

The following running time (real time) is recorded for **Insertion, Selection, Mergesort** and **Heapsort** based on the time taken to sort integers of various input sizes in CELL machine. All logs printed by each of the sorting algorithms are stored in a folder called **logs**.

Running time of different sorting algorithms with input size = 10

sorting algorithms /input files	10.1	10.2	10.3	10.4	10.5	Average
Insertion	0.004s	0.003s	0.005s	0.004s	0.004s	**0.004s**
Selection	0.002s	0.002s	0.002s	0.002s	0.002s	**0.002s**
Mergesort	0.002s	0.002s	0.002s	0.002s	0.002s	**0.002s**
Heapsort	0.002s	0.002s	0.002s	0.002s	0.002s	**0.002s**

Running time of different sorting algorithms with input size = 100

sorting algorithms/input files	100.1	100.2	100.3	100.4	100.5	Average
Insertion	0.004s	0.009s	0.004s	0.005s	0.005s	**0.0054s**
Selection	0.002s	0.002s	0.002s	0.002s	0.002s	**0.002s**
Mergesort	0.002s	0.002s	0.005s	0.002s	0.004s	**0.003s**
Heapsort	0.002s	0.002s	0.002s	0.002s	0.002s	**0.002s**

Running time of different sorting algorithms with input size = 1000

sorting algorithms/input files	1000.1	1000.2	1000.3	1000.4	1000.5	Average
Insertion	0.007s	0.006s	0.006s	0.006s	0.006s	**0.0062s**
Selection	0.005s	0.009s	0.005s	0.005s	0.005s	**0.0048s**
Mergesort	0.003s	0.003s	0.003s	0.003s	0.003s	**0.003s**
Heapsort	0.003s	0.003s	0.003s	0.003s	0.002s	**0.003s**

Running time of different sorting algorithms with input size = 10000

sorting algorithms/input files	10000.1	10000.2	10000.3	10000.4	10000.5	Average
Insertion	0.226s	0.203s	0.196s	0.201s	0.204s	**0.206s**
Selection	0.323s	0.286s	0.292s	0.273s	0.274s	**0.289s**
Mergesort	0.010s	0.010s	0.010s	0.010s	0.011s	**0.010s**
Heapsort	0.008s	0.008s	0.007s	0.009s	0.008s	**0.008s**

Running time of different sorting algorithms with input size = 100000

sorting algorithms/input files	100000.1	100000.2	100000.3	100000.4	100000.5	Average
Insertion	19.064s	19.380s	19.268s	19.186s	19.229s	**19.225s**
Selection	26.883s	26.955s	26.829s	27.285	26.567	**26.903s**
Mergesort	0.091s	0.100s	0.124s	0.113s	0.100s	**0.105s**
Heapsort	0.064s	0.089s	0.061s	0.060s	0.059s	**0.066s**

Running time of different sorting algorithms with input size = 1000000

sorting algorithms/input files	1000000.1	1000000.2	1000000.3	1000000.4	1000000.5	Average
Insertion	32m51.297s					
Selection	46m10.117s					

Mergesort	1.029s					
Heapsort	0.762s					

Brief Explanation of Sorting Algorithm

Insertion Sort

Insertion Sort is a simple comparison-based sorting algorithm that builds the sorted list one item at a time. The process involves picking an element and inserting it into its correct position within an already sorted sequence. Initially, the first element is considered a sorted sequence of size one. Then, the second element is inserted into its correct position, shifting the first element if necessary. This process continues until all elements are correctly positioned. While Insertion Sort performs well on small datasets, its **$O(n^2)$ time complexity** makes it inefficient for larger lists.

Selection Sort

Selection Sort takes a different approach by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and swapping it into its correct position. The algorithm first finds the smallest element and places it at the beginning. Then, it finds the next smallest element and places it in the second position, and so on, until the entire list is sorted. This sorting method also runs in **$O(n^2)$ time**, making it impractical for large datasets despite its straightforward implementation.

Merge Sort

Merge Sort is a **divide-and-conquer** algorithm that splits the input list into two halves, recursively sorts each half, and then merges them back together in sorted order. The key advantage of Merge Sort is that it maintains **O(n log n) time complexity**, making it efficient even for large datasets. The merging step ensures that the final list is fully sorted after all recursive divisions have been resolved. Unlike Insertion and Selection Sort, Merge Sort performs significantly better on large inputs.

Heapsort

Heapsort is another efficient **O(n log n) sorting algorithm** that utilizes a heap data structure to organize and extract elements in sorted order. It first transforms the input into a **min-heap** (or max-heap) and then extracts elements one by one to form the sorted list. Unlike Merge Sort, which requires additional memory for merging, Heapsort sorts the elements in-place, making it an excellent choice for memory-constrained environments.

Summary of Order of Growth

From the recorded execution times, we can observe the following:

- **Insertion Sort and Selection Sort show $O(n^2)$ behavior**, making them impractical for large datasets. For instance, sorting 1,000,000 elements took **32 minutes and 51 seconds for Insertion Sort**, while Selection Sort required **46 minutes and 10 seconds**.
- **Merge Sort and Heapsort consistently run in O(n log n) time**, proving to be far more efficient. For 1,000,000 elements, Merge Sort completed in just **1.029 seconds**, while Heapsort was even faster at **0.762 seconds**.
- **For small inputs (≤1000), the execution time differences are negligible**, with all sorting algorithms completing in milliseconds. However, as input size grows, the impact of algorithmic efficiency becomes significantly more pronounced.
- The **gap between $O(n^2)$ and O(n log n) algorithms grows exponentially** as input size increases, reinforcing why **Merge Sort and Heapsort should always be preferred for large datasets** compared to Insertion sort and Selection sort.

Sorting Algorithms Execution Guide – CELL Machine

This document provides instructions on how to manually execute sorting algorithms or use an automated script for batch execution.

Manual Execution

You can manually compile and execute any sorting algorithm by following these steps:

1. Navigate to the Sorting Algorithm Folder

Move into the directory of the sorting algorithm you want to test. For example, to run Heapsort:

$ cd Heapsort

2. Compile the C++ Code

Use g++ to compile all *.cpp* files in the folder:

$ g++ *.cpp -o a.out

3. Execute the Program with an Input File

Replace * with the specific input file name located in the *./inputs/* folder.

$ ./a.out ./inputs/*

Example:

$ ./a.out ./inputs/input.100000.1

4. Measure Execution Time

To measure the execution time of a specific input file, use the `time` command. Replace * with the specific input file name located in the `./inputs/` folder.

$ time for f in ./inputs/*; do echo $f; ./a.out $f; done

Example:

$ time for f in ./inputs/input.100000.1; do echo $f; ./a.out $f; done

Automated Execution (Recommended)

Each sorting algorithm folder contains an automation script called `run_tests.sh`, which allows us to efficiently run tests on multiple input files.

1. Navigate to the Sorting Algorithm Folder

Example:

$ cd Heapsort

2. Grant Execution Permission to the Script

Before running the script, ensure it has the correct execution permissions:

$ chmod +x run_tests.sh

3. Run the Script

Execute the script to automatically run the sorting algorithm on multiple input files:

$ ./run_tests.sh

4. Expected Behavior

- The script compiles the C++ program.
- It runs the program for all available input files in the *`./inputs/`* directory.
- It displays execution time for each input file.
- It provides a summary of the total number of files processed.

Additional Notes

- Ensure that all input files are stored in the *`./inputs/`* directory.
- The script automatically skips non-input files (e.g., `generate_input.py`).
- If an error occurs during compilation, execution will be halted.