**Design and Analysis of Algorithms**

**Performance Analysis of Sorting Algorithms**

By:

**A R Sharan Kumar** (231CS101)

**Yashwanth R** (231CS265)

To:

**Vani M Associate Professor**

****

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA, SURATHKAL-575025**

**1: COMPARISON OF SORTING ALGORITHMS**

**Sorting Algorithms: Theoretical and Experimental Analysis**

Sorting algorithms are assessed in two main ways: **theoretically** and **experimentally**.

1. **Theoretical Analysis**: This involves studying the algorithm's efficiency using time complexity, which helps determine how it performs in **best-case, worst-case, and average-case** scenarios. It gives a mathematical understanding of how the algorithm scales as the input size increases.
2. **Experimental Evaluation**: This means running the algorithm on actual datasets to measure its real execution time. The actual speed can vary depending on factors such as the **nature of the input data (random, sorted, or reversed), the efficiency of the hardware used, and how well the algorithm is implemented** in a programming language.

While theoretical analysis helps predict performance trends, real-world testing provides practical insights into how an algorithm behaves under different conditions.

**1. Bubble Sort**

**Theoretical Complexity:**

* Best Case: O(n) (When the array is already sorted, requiring just one pass)
* Average Case: O(n²)
* Worst Case: O(n²) (When the array is in reverse order)

**Practical Performance:**

* Very inefficient for large datasets due to its slow quadratic growth.
* Performs worse than most other sorting algorithms.
* Works best for small or nearly sorted data because of its simplicity.

**2. Insertion Sort**

**Theoretical Complexity:**

* Best Case: O(n) (Already sorted array)
* Average Case: O(n²)
* Worst Case: O(n²) (Reverse sorted array)

**Practical Performance:**

* More efficient than Bubble Sort for small datasets.
* Performs well when the data is nearly sorted because it minimizes swaps.

**3. Merge Sort**

**Theoretical Complexity:**

* Best Case: O(n log n)
* Average Case: O(n log n)
* Worst Case: O(n log n)

**Practical Performance:**

* Offers stable performance across different types of input.
* Uses extra memory since it needs additional space for merging, meaning it is not an in-place sort.
* Commonly used for external sorting and cases where stability is required**.**

**4. Quick Sort**

**Theoretical Complexity:**

* Best Case: O(n log n) (Occurs when partitions are balanced)
* Average Case: O(n log n)
* Worst Case: O(n²) (Happens when partitions are unbalanced, such as when sorting an already sorted array with a poor pivot choice)

**Practical Performance:**

* One of the fastest sorting algorithms in practice, often outperforming Merge Sort.
* In-place sorting reduces memory usage.
* Choosing a good pivot (e.g., random pivot) is crucial for better performance.

**5. Heap Sort**

**Theoretical Complexity:**

* Best Case: O(n log n)
* Average Case: O(n log n)
* Worst Case: O(n log n)

**Practical Performance:**

* Slower than Quick Sort due to poor cache efficiency.
* Useful for priority queues, where quick removal of the largest (or smallest) element is needed.
* Works in-place but does not maintain the original order of equal elements (not stable).

**6. Radix Sort**

**Theoretical Complexity:**

* Best Case: O(nk)
* Average Case: O(nk)
* Worst Case: O(nk)  
  *(where k is the number of digits in the largest number)*

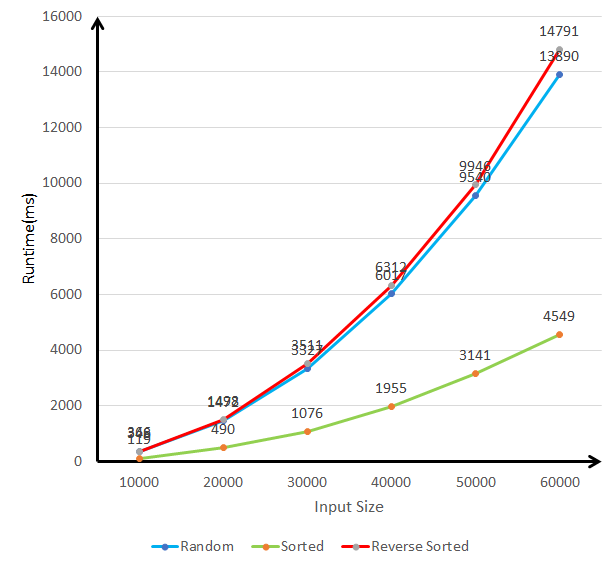
**Practical Performance:**

* Faster than comparison-based sorting when handling numbers with a limited range of digits.
* Works only for numeric data or fixed-length strings.
* Requires additional memory, making it unsuitable for cases with high space constraints.

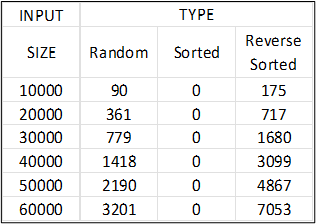
**Graphs of all Sorting Algorithms (runtime vs input size):**

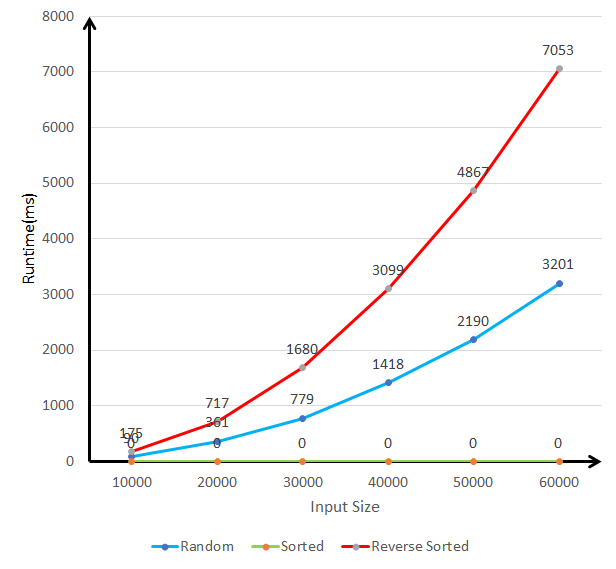
1. **Bubble Sort:**

****

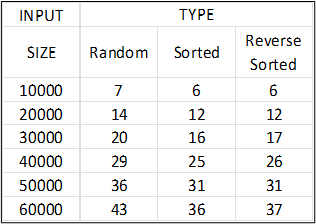
****

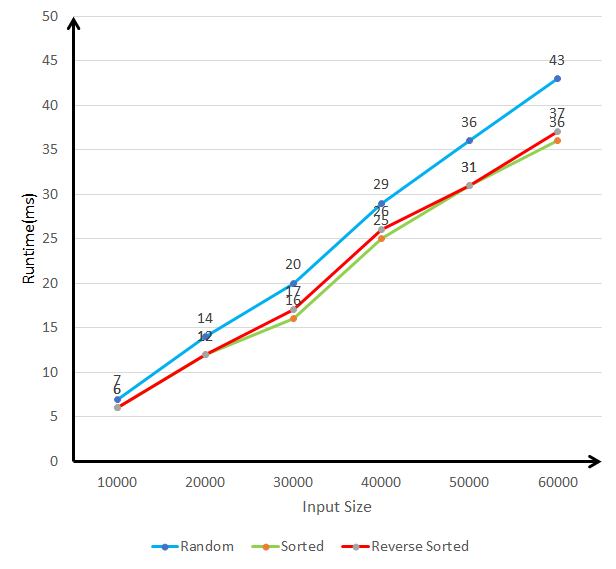
1. **Insertion Sort:**

****

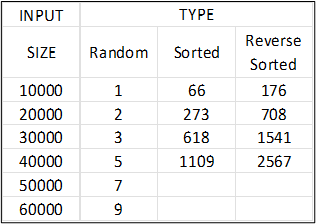
****

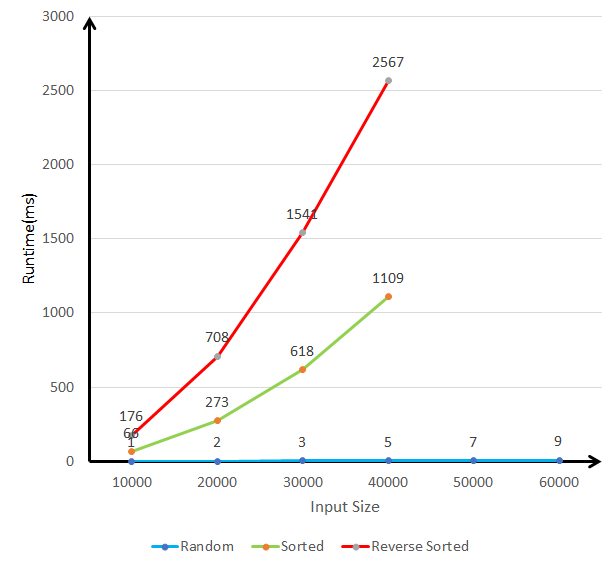
1. **Merge Sort:**

****

****

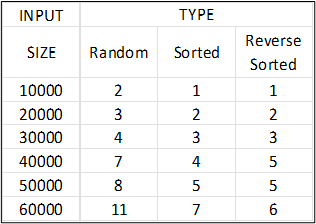
1. **Quick Sort: First Element as Pivot:**

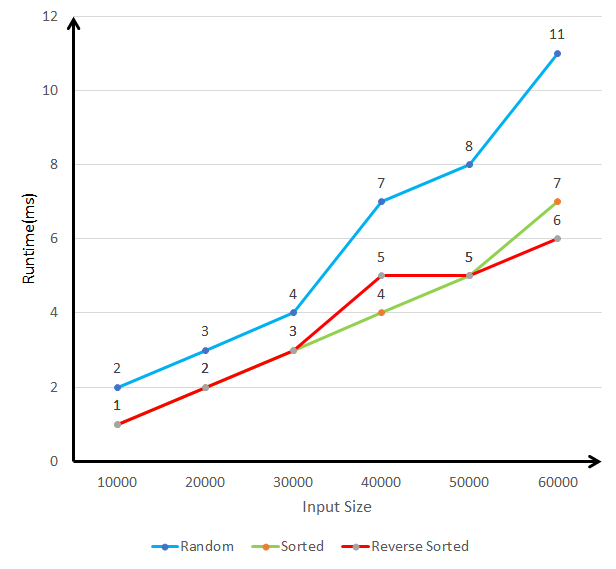
****

****

Note: Outputs were not obtained for sorted and reverse sorted inputs of size 50,000 and 60,000, most likely due to stack overflow.

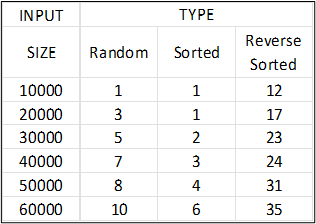
1. **Quick Sort: Random Element as Pivot:**

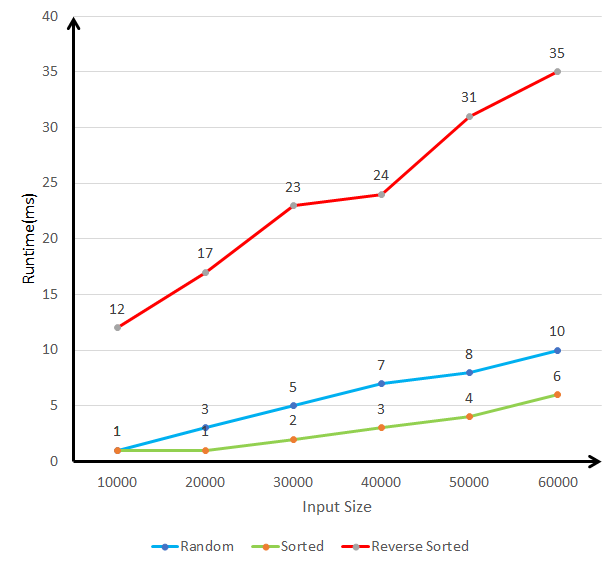
****

****

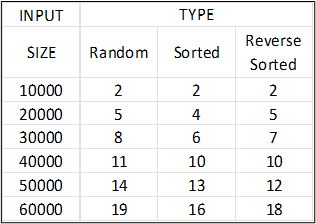
1. **Quick Sort: Median of first, middle and last elements of the array:**

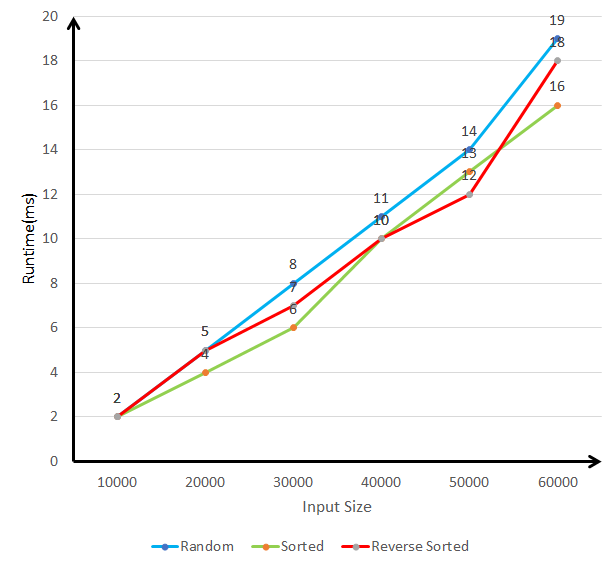
Quick Sort: Median of Low, High, and Mid



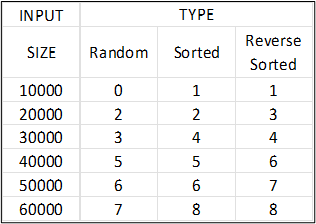


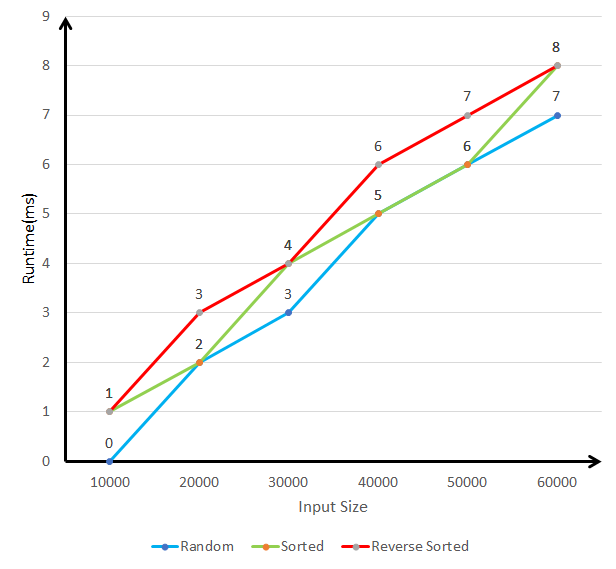
1. **Heap Sort:**





1. **Radix Sort:**





**2: DATA GENERATION AND EXPERIMENTAL SETUP**

* **What kind of machine did you use?**

Machine specifications:

* •Processor: 13th Gen Intel® Core™ i5 1335U
* •Cores: 10
* •L1 Cache: 928 KB
* •L2 Cache: 6.5 MB
* •L3 Cache: 12.0 MB
* •Memory: 16 GB
* OS: Microsoft Windows 11 Home
* **What timing mechanism?**

The program measures the execution time using the **Chrono library** in C++. Below is a detailed breakdown of the timing mechanism:

**1. Library Used: chrono**

The chrono library is part of the C++ Standard Library (<chrono>), which provides **high-precision time utilities** for measuring time intervals.

**2. Capturing Start Time**

auto start = high\_resolution\_clock::now();

* high\_resolution\_clock::now() is used to get the current timestamp before sorting begins.
* high\_resolution\_clock is the **most precise clock available** in <chrono>, often mapped to:
  + **std::steady\_clock** (if the system supports a steady clock)
  + **std::system\_clock** (if steady clock isn't available)

**3. Capturing End Time**

auto stop = high\_resolution\_clock::now();

* This takes another timestamp **right after** Radix Sort finishes executing.

**4. Calculating the Duration**

auto duration = duration\_cast<milliseconds>(stop - start);

* (stop - start) computes the **time difference** between start and stop.
* duration\_cast<milliseconds> converts the time duration into **milliseconds** (ms).
* milliseconds is a unit of time where:
  + **1 millisecond = 1/1000th of a second (0.001s)**

**5. Displaying the Execution Time**

cout << "Time taken to sort: " << duration.count() << " milliseconds" << endl;

* .count() extracts the numerical value (in ms).
* The result is printed on the console.
* **How many times did you repeat each experiment?**

Every input for each algorithm was executed 5 times and noted down manually.

* **What times are reported?**

The code gives the corresponding runtime for the input file opened. The output time reported is in milliseconds.

* **How did you select the inputs?**

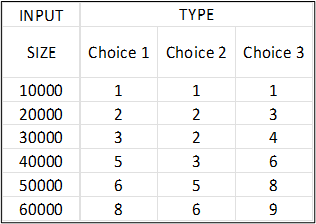
inputs were generated by LLMs, since generation of 60,000 integers is not manually feasible. However, the input data is cross checked by the LLM for sorted and reverse sorted arrays. There are three types of arrays used for plotting the graphs:

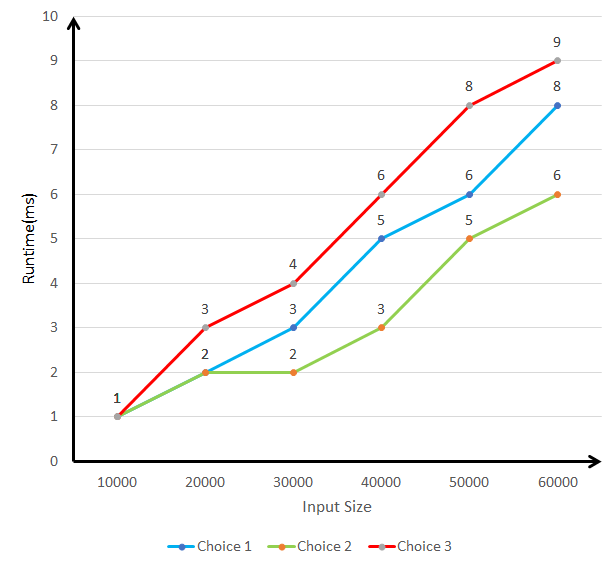
* + Randomly generated array.
  + Randomly generated array sorted in ascending order.
  + Randomly generated array sorted in descending order.
* **Did you use the same inputs for all sorting algorithms?**

Yes, the same input array is used for all sorting algorithms in each experiment. This ensures that all algorithms are tested on identical inputs for a fair comparison.

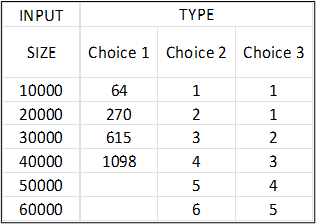
**3: QUICK SORT VERSIONS COMPARISON**

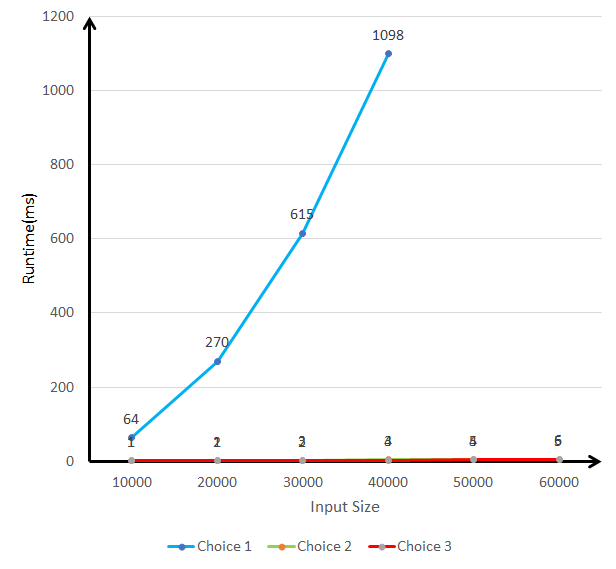
* **Graph the best case running time as a function of input size n for all three versions (use the best case input you determined in each case in part 1).**
* **Random Input**

****

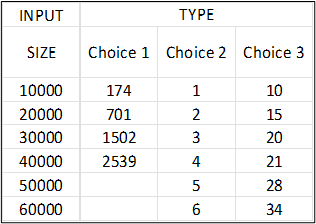
****

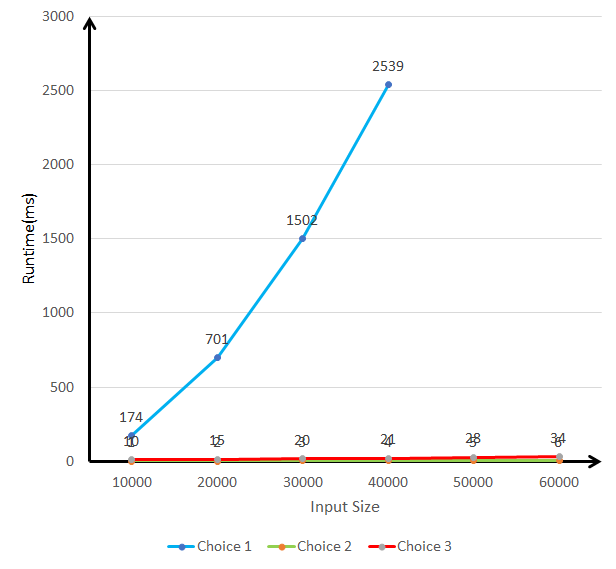
* **Sorted Input**



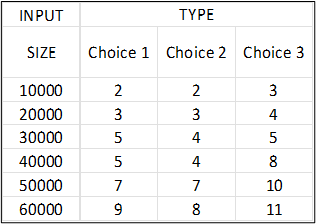


* **Reverse Sorted Input**

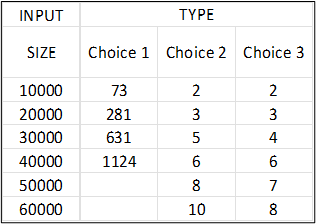
****

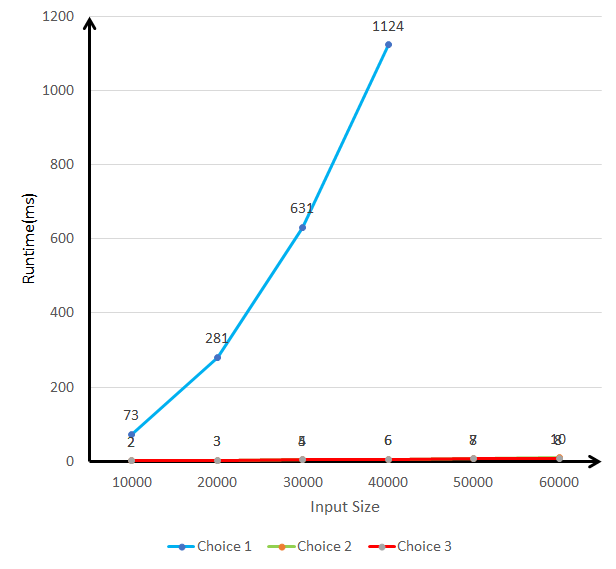
****

* **Graph the worst case running time as a function of input size n for all three versions (use the worst case input you determined in each case in part 1).**
* **Random Input**

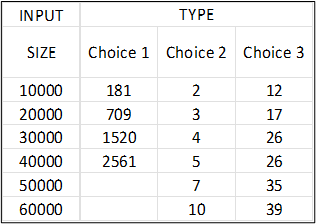
****

* **Sorted Input**

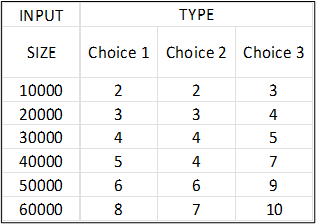
****

****

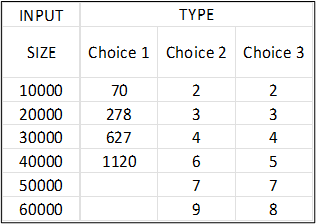
* **Reverse Sorted Input**

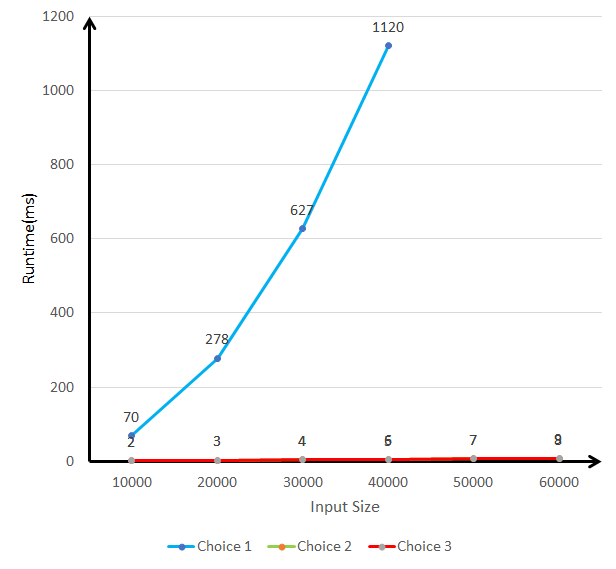
****

* **Graph the average case running time as a function of input size n for all three versions.**
* **Random Input**

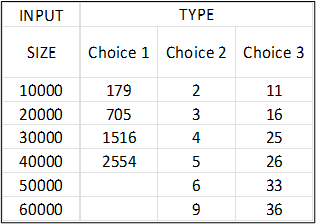
****

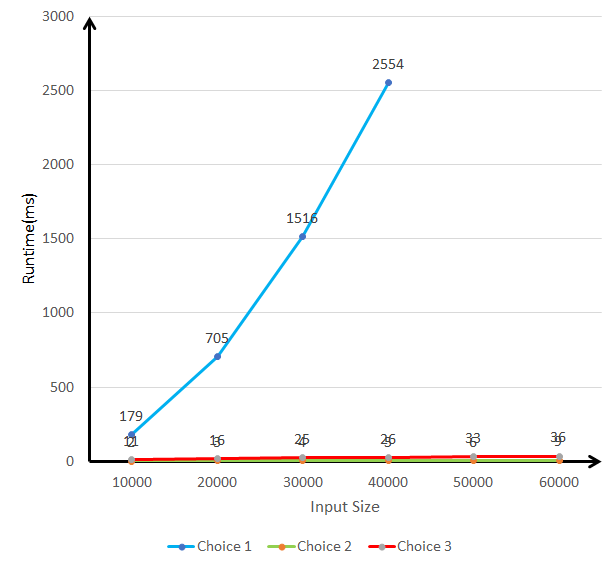
* **Sorted Input**

****

****

* **Reverse Sorted Input**

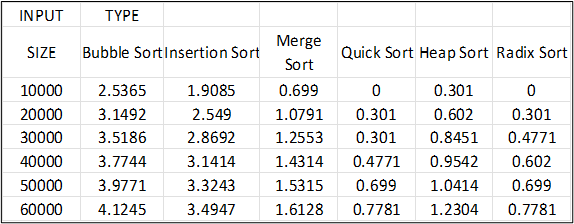
****

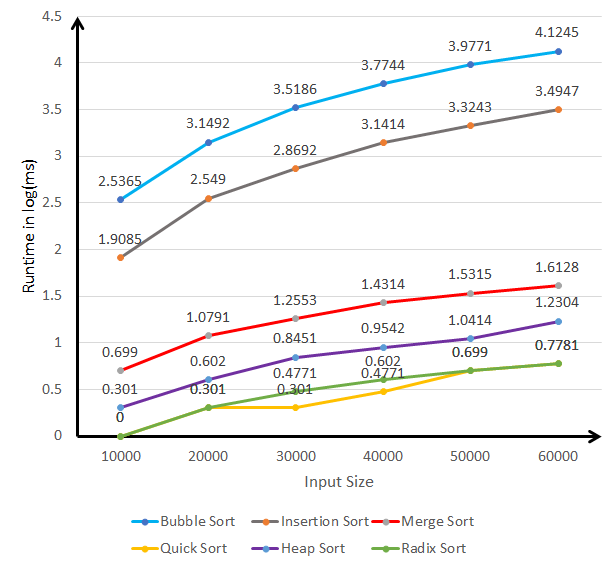
****

From the graphs, we can conclude that choosing random element as pivot outperformed the other two choices in the Quick Sort Algorithm for our inputs.

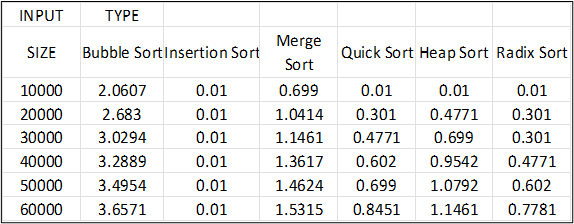
**4: ALL ALGORITHMS COMPARISON**

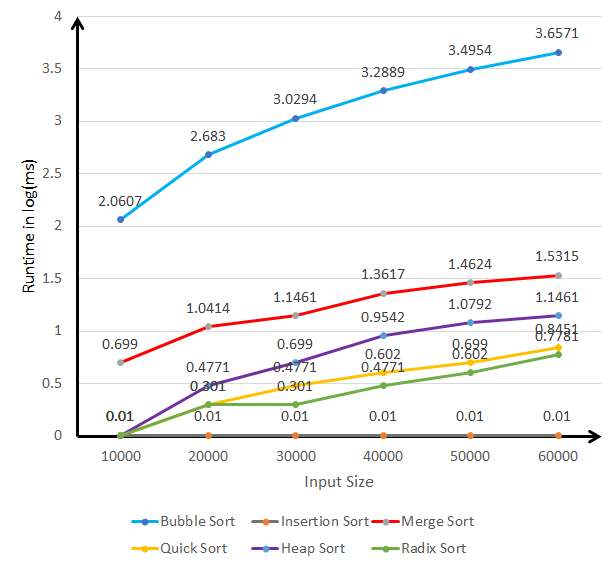
* Graph the best-case running time as a function of input size n for the six sorts
* Random Input



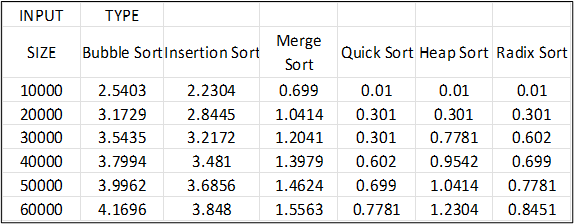


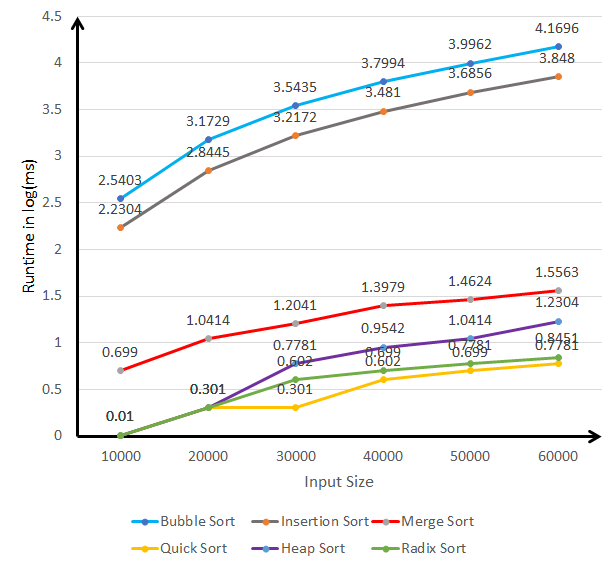
* Sorted Input



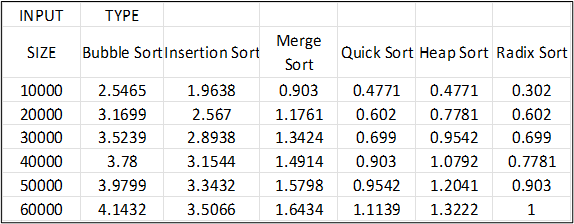


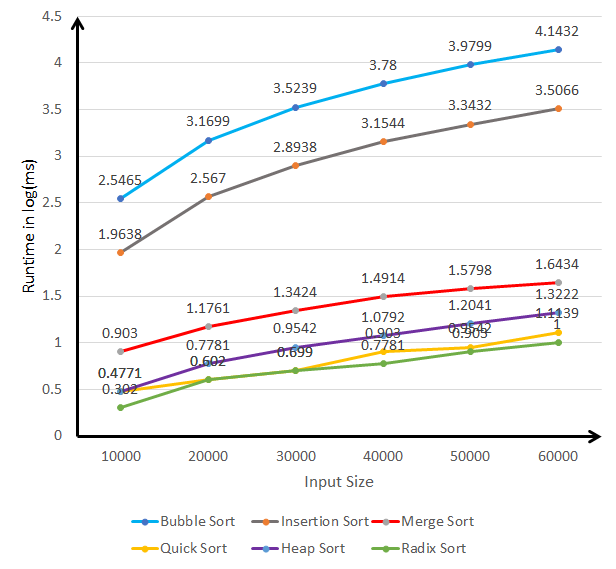
* Reverse Sorted Input



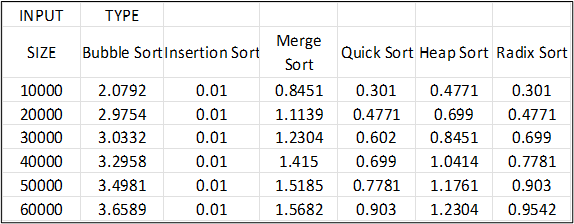


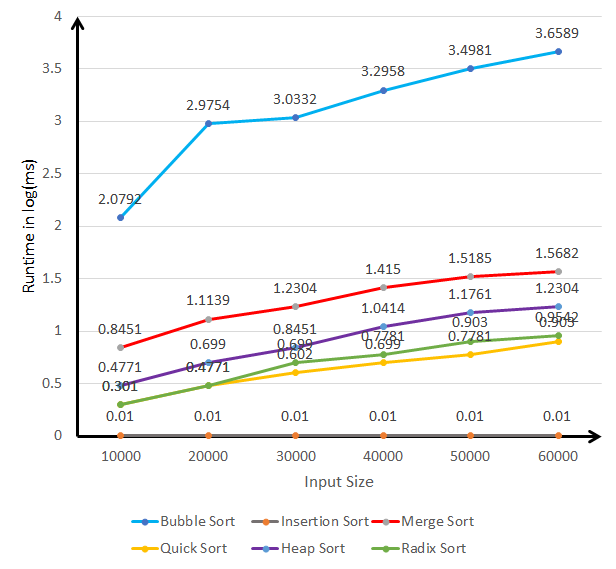
* Graph the worst-case running time as a function of input size n for the six sorts
* Random Input



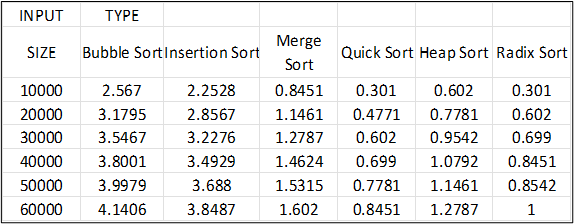


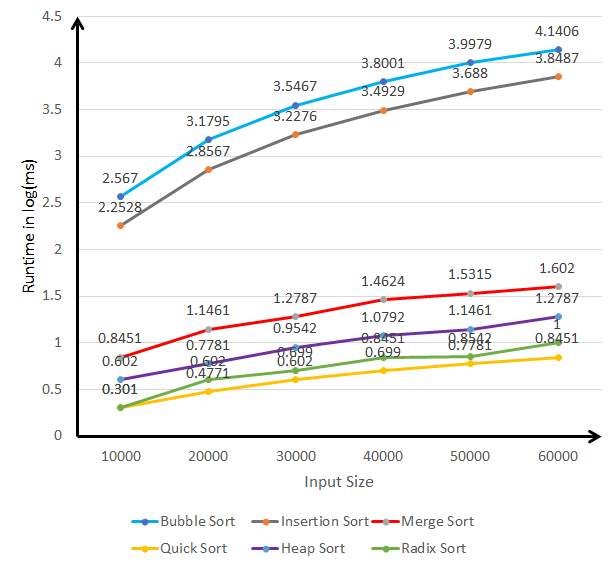
* Sorted Input



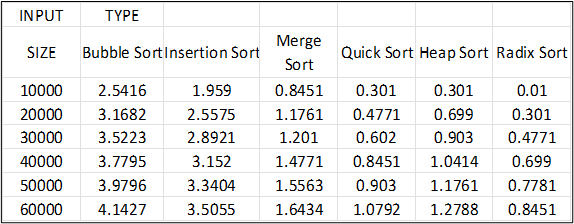


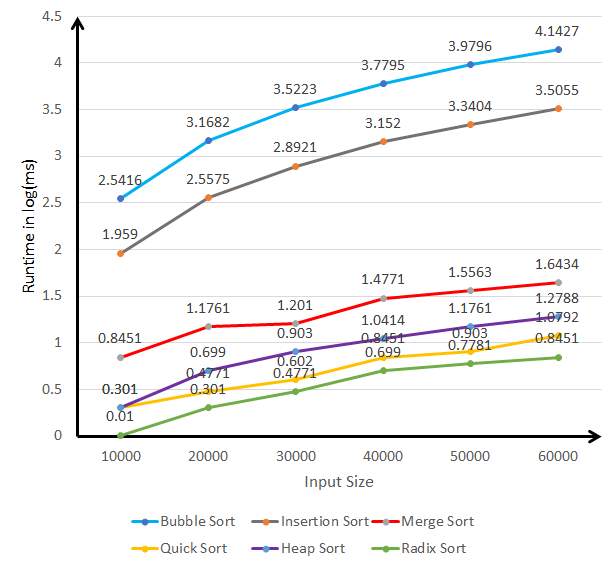
* Reverse Sorted Input



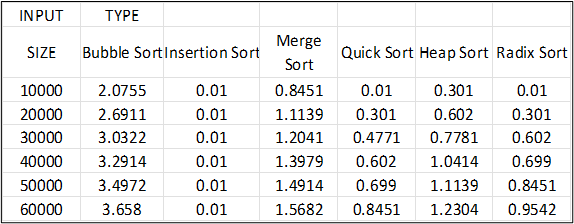


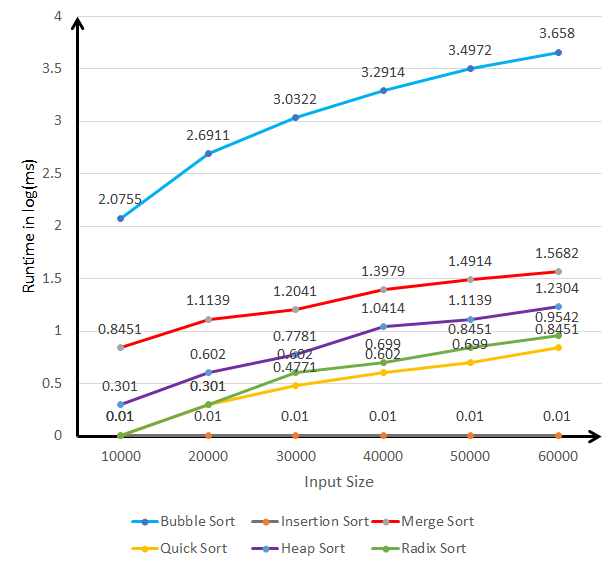
* Graph the average case running time as a function of input size n for the six sorts.
* Random Input



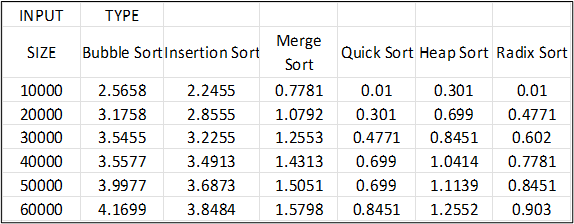


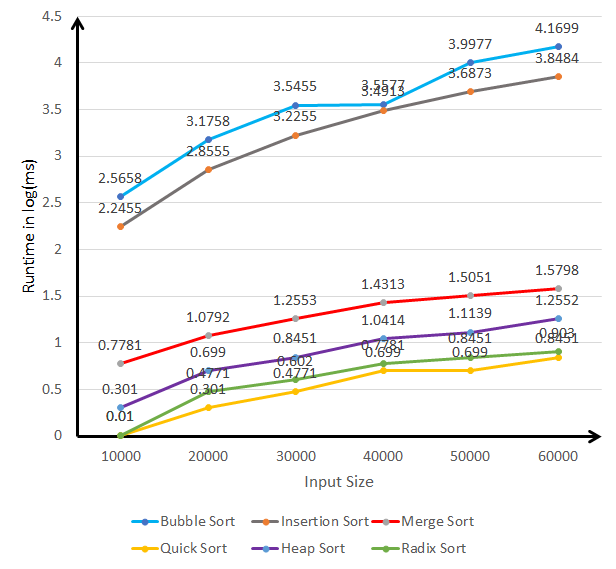
* Sorted Input





* Reverse Sorted Input





**5: Analysis of Data**

* The above graphs show that the runtime is directly proportional to the input size. Since the number of comparisons increase with input size, we can conclude that number of comparisons is correlated with execution time. The graphs are already plotted above in the 1st main.