COURSE CODE	:	CZ2001
COURSE NAME	:	ALGORITHMS
GROUP MEMBERS	:	

1. **Alvin Lee Yong Teck** **(U1620768F)**
2. **Htet Naing** **(U1620683D)**
3. **Liu Jihao, Bryan** **(U1621401C)**
4. **Lim Kian Hock, Bryan** **(U1620949L)**
5. **Luke Lee** **(U1620850A)**
6. **Zacharias Tay** **(U1620760H)**

LAB GROUP	:	SS4
COURSEWORK	:	EXAMPLE CLASS 3 REPORT
TITLE	:	INTEGRATION OF MERGE SORT & INSERTION SORT

Contents

Introduction to Integration of Merge Sort and Insertion Sort

MergeSort is a divide-and-conquer algorithm. MergeSort first divides the problem into sub-problems and continues to divide sub-problems into smaller sub-problems until no more sub-problems can be divided any further. Since our goal is to sort the items in a given list, after dividing the original problem into many small sub-problems, we have to conquer our sub-problems by sorting the items in each problem recursively.

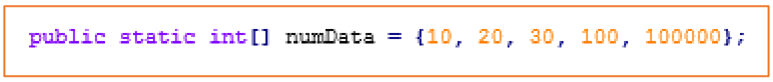
When the size of a sub-problem is relatively small, the overheads incurred by invoking many recursive methods render the MergeSort inefficient. Such inefficiency can be alleviated with the assistance of insertion sort, a sorting algorithm that runs efficiently for relatively small input. A small suitable threshold value (**S)** will be selected for limiting the size of a sub-problem. Whenever the sub-problem size reaches the threshold **(S)**, the algorithm will invoke insertion sort instead of proceeding to divide smaller sub-problems (as in the original MergeSort). The pseudo code for the modified MergeSort can be seen below:

```
void mergeSort(Element E[], int first, int last, int S)
{
     if (last - first > S) {
          int mid = (first + last)/2;
          mergeSort(E, first, mid, S);
          mergeSort(E, mid + 1, last, S);
          merge(E, first, mid, last);
     } else {
          insertionSort(E, first, last);
     }
}
```

Figure 1. Mergesort pseudo code

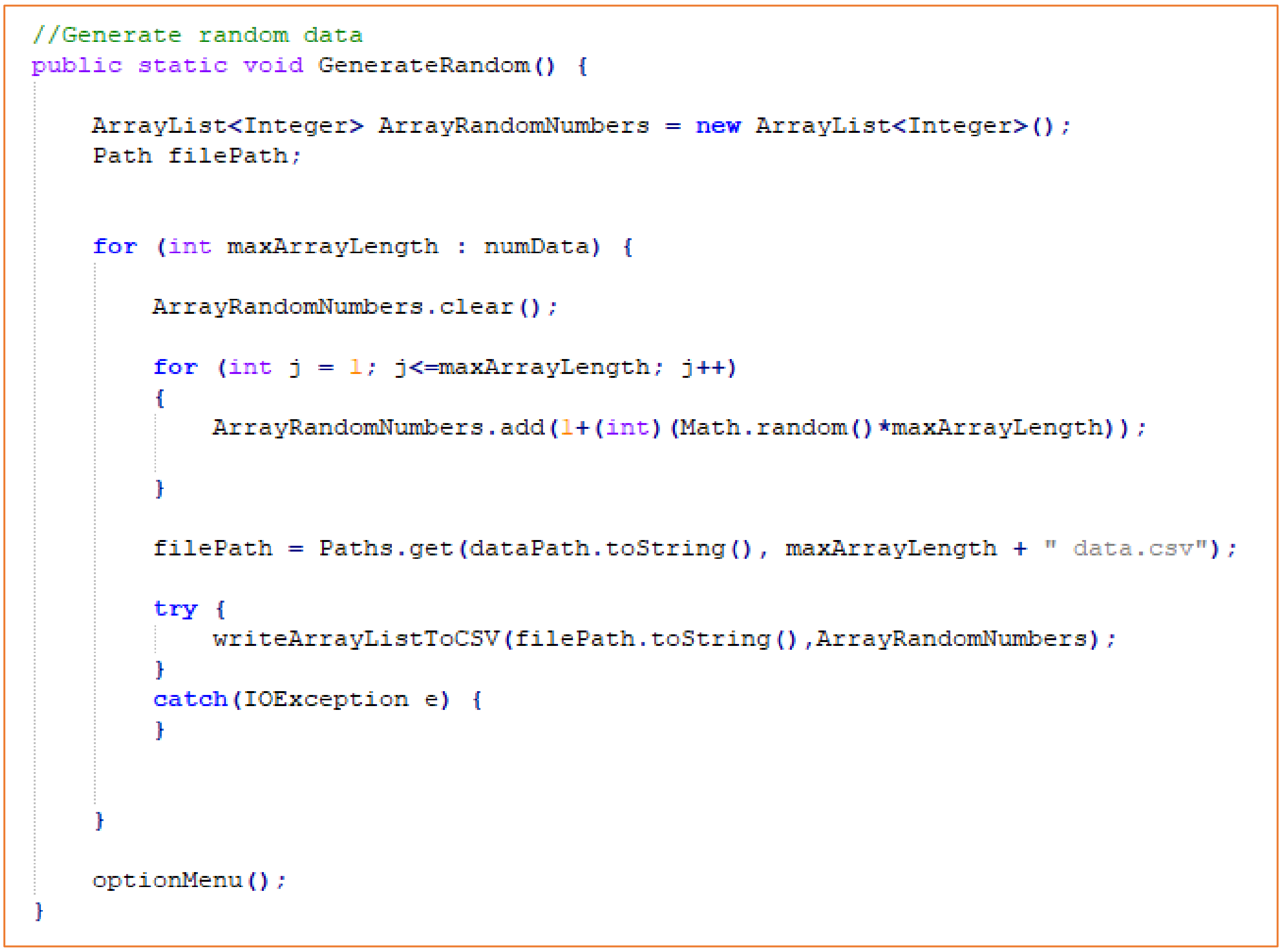
Generating Various Sizes of Input Data Based on Some Random Integers

The following line of code defines “numData” which is an int array that holds different input sizes.



```
public static int[] numData = {10, 20, 30, 100, 100000};
```

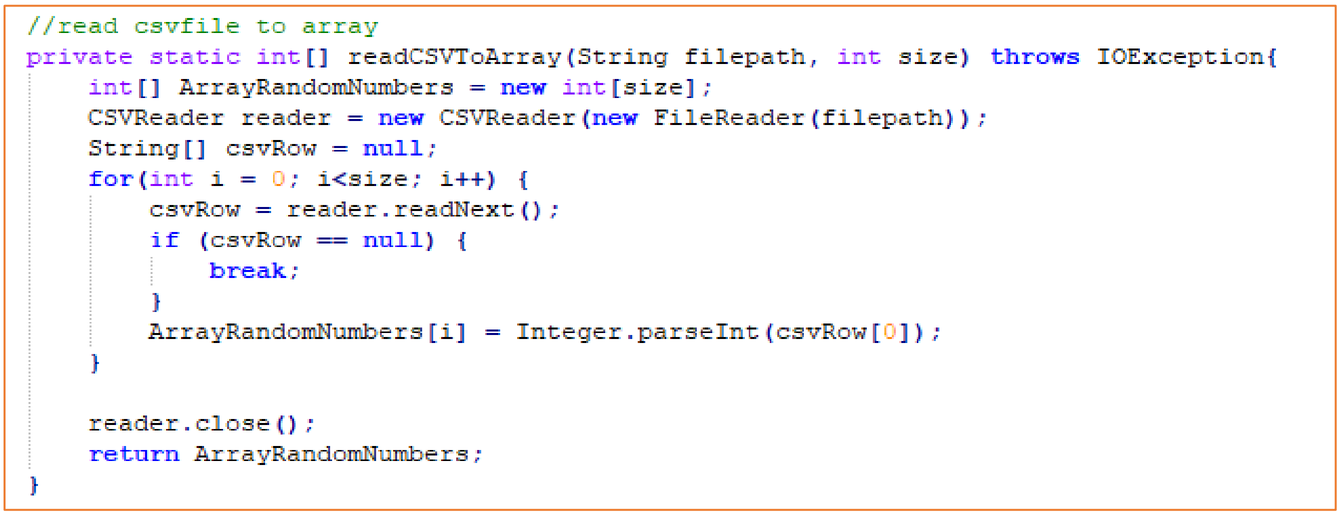
Therefore, our first array will have a length of 10 and will hold random integers ranging from “1 to 10”. The second array will have a length of 20 and will include integers ranging from “1 to 20” and so on. Eventually, our last array with a length of 100000 will be populated with random integers that vary from 1 to 100000.



```
//Generate random data
public static void GenerateRandom() {

    ArrayList<Integer> ArrayRandomNumbers = new ArrayList<Integer>();
    Path filePath;


    for (int maxArrayLength : numData) {

        ArrayRandomNumbers.clear();

        for (int j = 1; j<=maxArrayLength; j++)
        {
            ArrayRandomNumbers.add(1+(int)(Math.random()*maxArrayLength));

        }

        filePath = Paths.get(dataPath.toString(), maxArrayLength + " data.csv");

        try {
            writeArrayListToCSV(filePath.toString(),ArrayRandomNumbers);
        }
        catch(IOException e) {
        }


    }

    optionMenu();
}
```

Figure 2. A method that generates some random integers and saves them inside a CSV file

After that, the following code helps retrieve our data from the CSV file and assign them into respective arrays for sorting later.



```java
//read csvfile to array
private static int[] readCSVToArray(String filepath, int size) throws IOException{
    int[] ArrayRandomNumbers = new int[size];
    CSVReader reader = new CSVReader(new FileReader(filepath));
    String[] csvRow = null;
    for(int i = 0; i<size; i++) {
        csvRow = reader.readNext();
        if (csvRow == null) {
            break;
        }
        ArrayRandomNumbers[i] = Integer.parseInt(csvRow[0]);
    }

    reader.close();
    return ArrayRandomNumbers;
}
```

Figure 3. A method that reads randomly generated integers and assign them to respective arrays

Monitoring Running Times between the Two Versions of MergeSort

The running time of original MergeSort runs a lot slower than the modified version (MergeSort + InsertionSort). This is because once it reaches the threshold, it uses insertion sort which is efficient for sorting small data. (e.g. 10) as opposed to using MergeSort which is only efficient for sorting large data. This will reduce the overall running time of the modified version, which allows it to sort data faster than the original version. If we have high threshold (e.g. 1000), it will cause large data (e.g. 1000) to sort less efficiently compared solely by using InsertionSort.

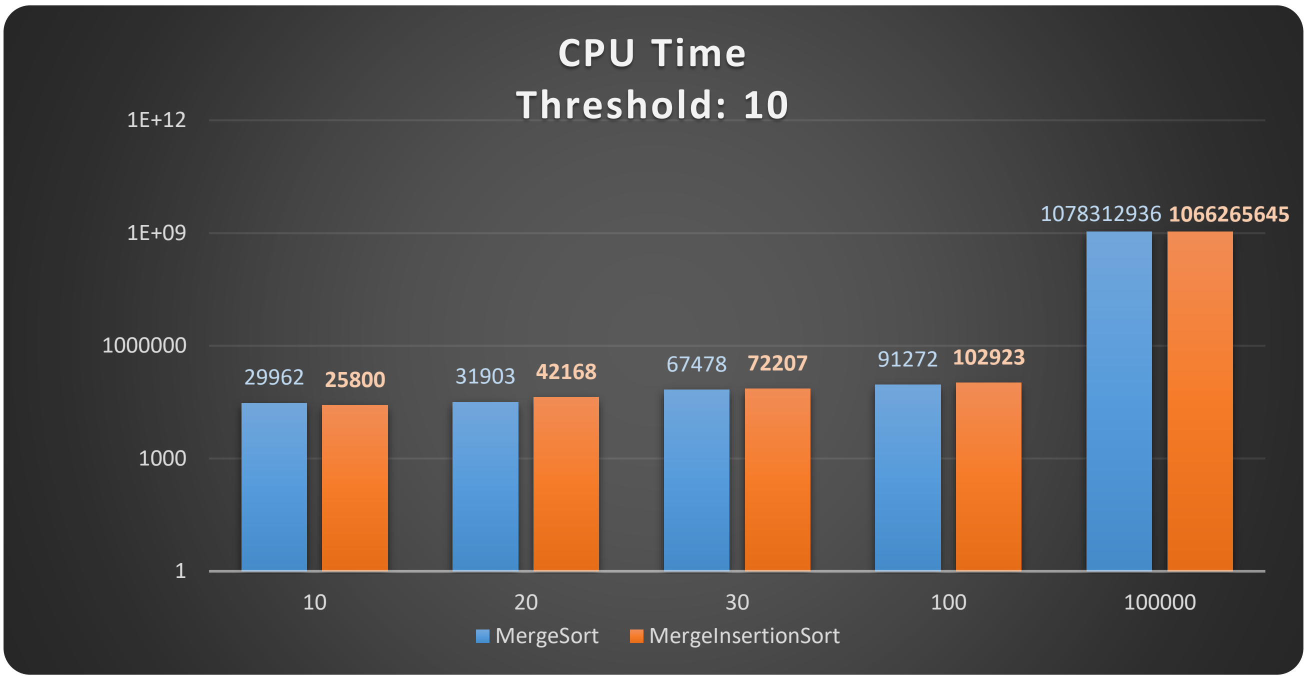


Figure 4. CPU running times in nano secs between MergeSort vs MergeInsertionSort with respect to different input sizes (Threshold = 10)

Monitoring Key Comparisons between the Two Versions of MergeSort

The number of comparison on average for original MergeSort is a lot smaller than the modified version across all the threshold. This is because when the data reaches a certain threshold, it uses InsertionSort to sort the data. Since InsertionSort does more comparison compared to MergeSort, this causes the modified version to have a lot more comparison compared to the original one. However, it is important to note that despite the modified version having more comparison on average, it performs a lot better than the original version in terms of running time.

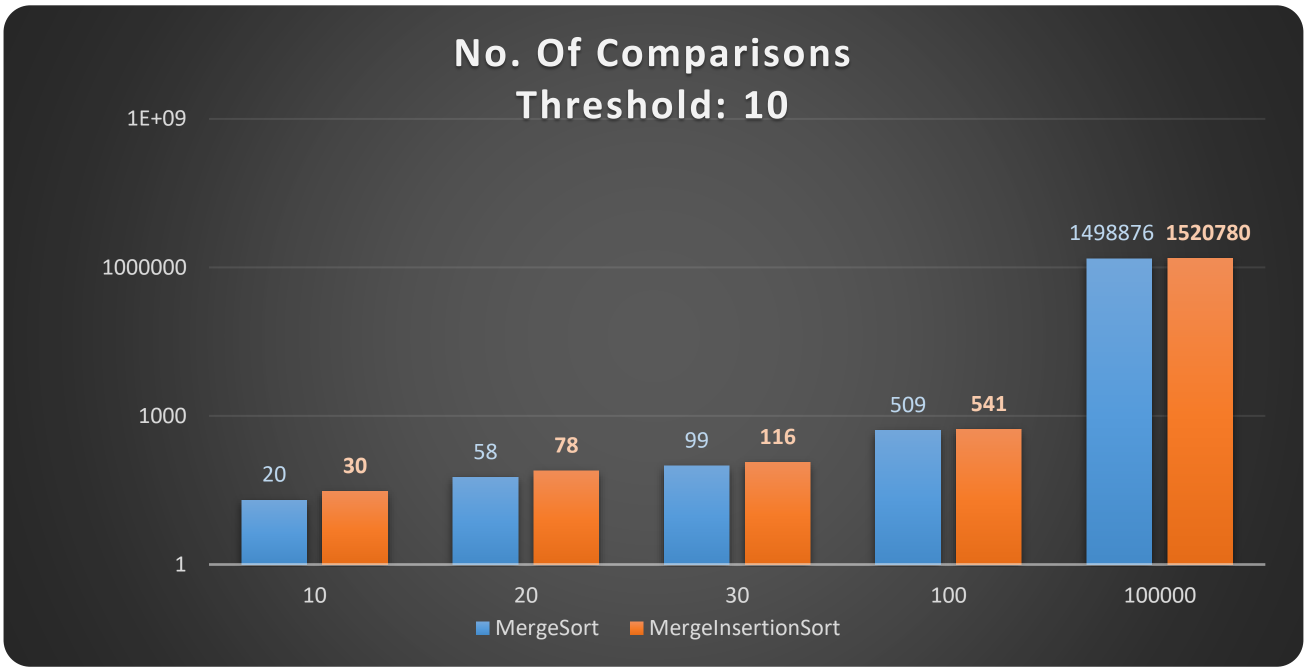


Figure 5. Number of key comparisons between MergeSort vs MergeInsertionSort with respect to different input sizes (Threshold = 10)

Performance of the Modified Algorithms with Respect to Different Threshold Values

Since our main goal is to be able to sort large data efficiently, we will only focus on analysing large input (e.g. 100000) to rationalize our charts below. In Figure 6, when our threshold value is 10, the running time for MergeInsertionSort is faster than MergeSort. However, when we use higher threshold values, the running time for MergeInsertionSort becomes slower as compared to threshold 10. Furthermore, since threshold is equivalent to number of items to be sorted, InsertionSort will have to sort more items and it causes the CPU running time to be less efficient. Therefore, in order to get the optimal performance when using MergeInsertionSort, it is always recommended to select the lower threshold value (e.g. 10 in our case).

Threshold: 10

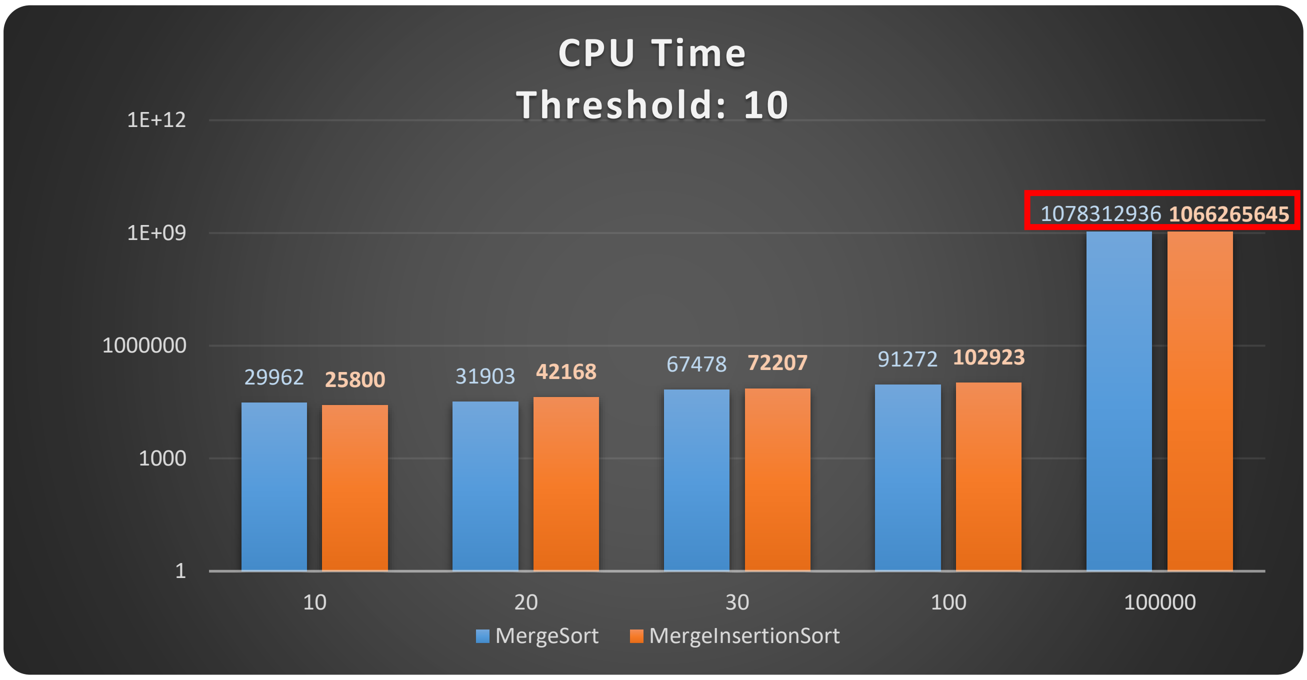


Figure 6. CPU running times in nano secs between MergeSort vs MergeInsertionSort with respect to different input sizes (Threshold = 10)

Threshold: 20

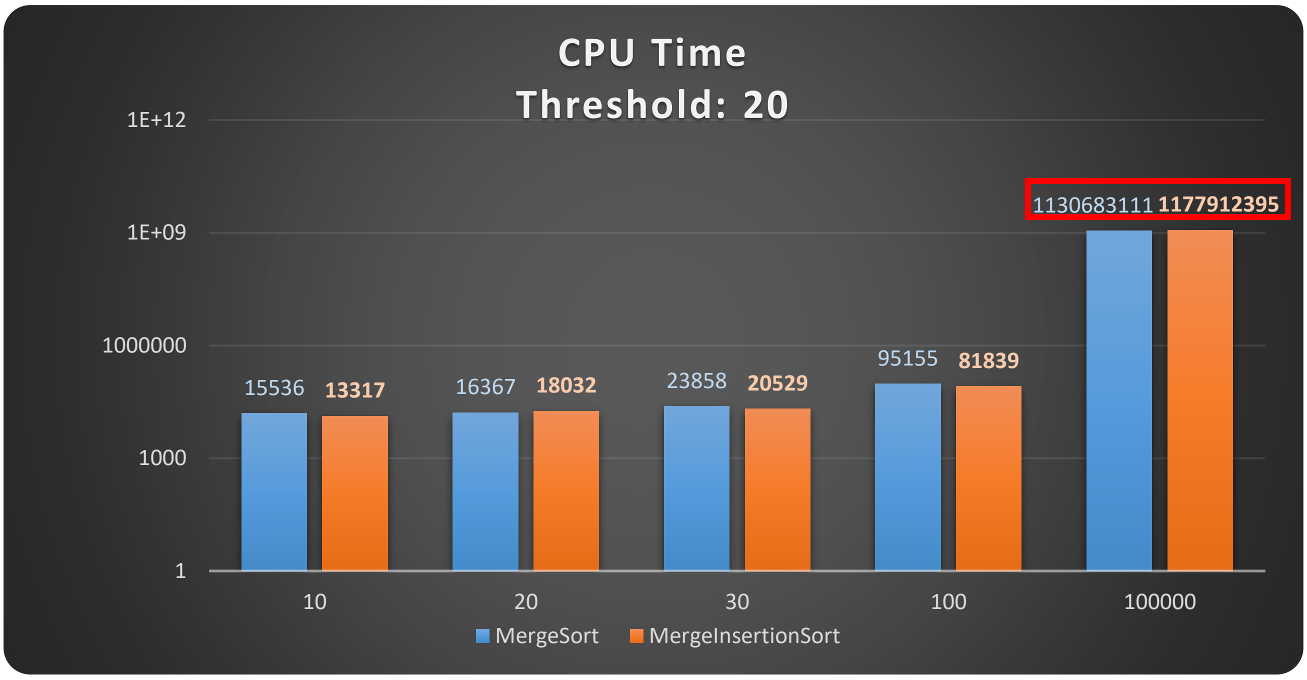


Figure 7. CPU running times in nano secs between MergeSort vs MergeInsertionSort with respect to different input sizes (Threshold = 20)

Threshold: 30

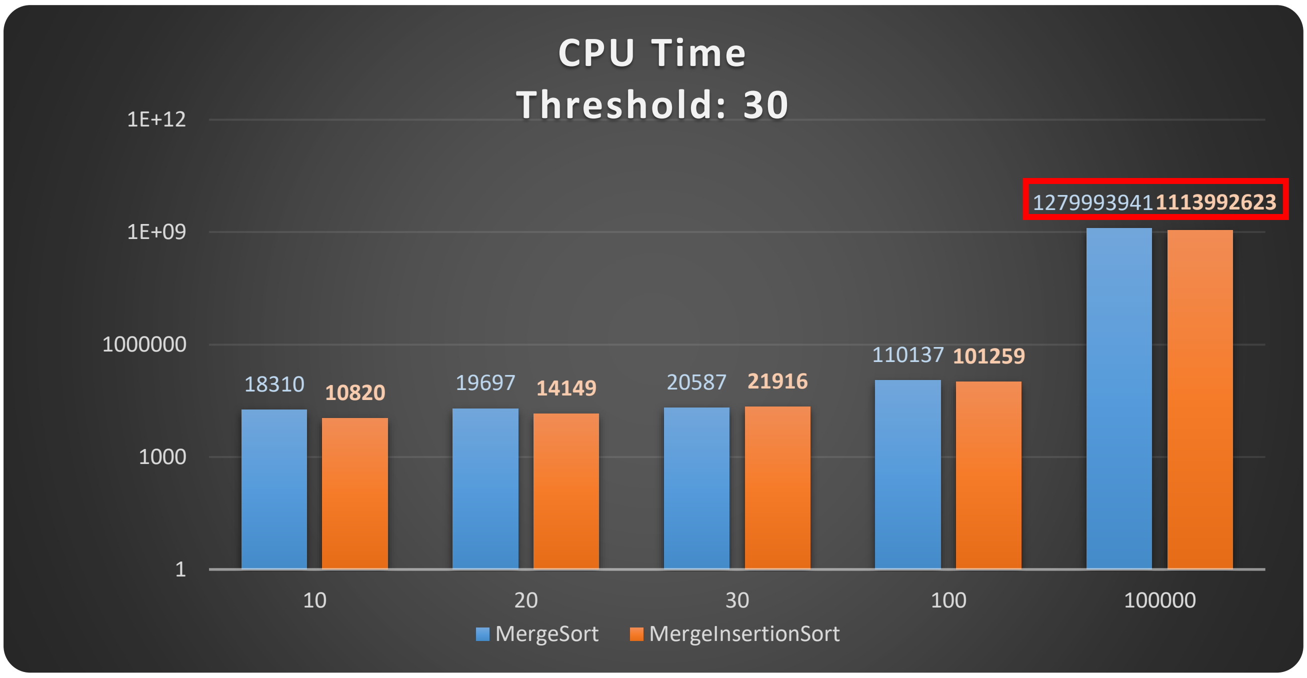


Figure 8. CPU running times in nano secs between MergeSort vs MergeInsertionSort with respect to different input sizes (Threshold = 30)

Threshold: 60

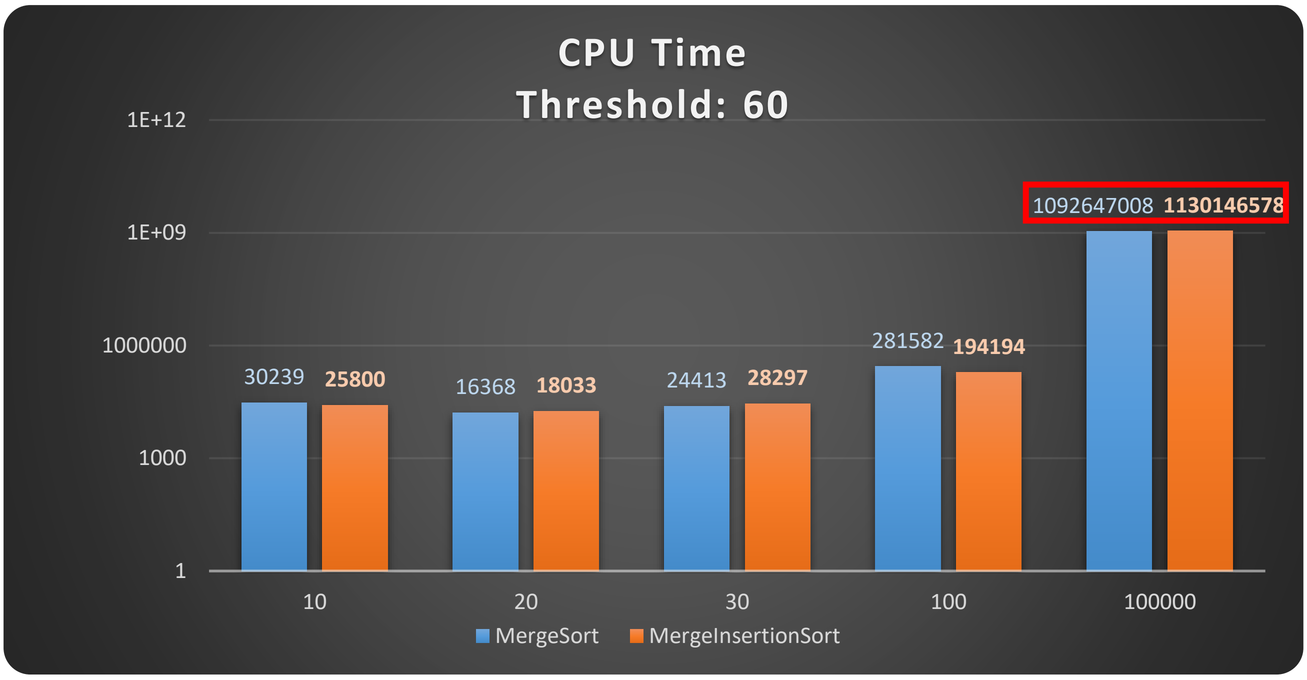


Figure 9. CPU running times in nano secs between MergeSort vs MergeInsertionSort with respect to different input sizes (Threshold = 60)