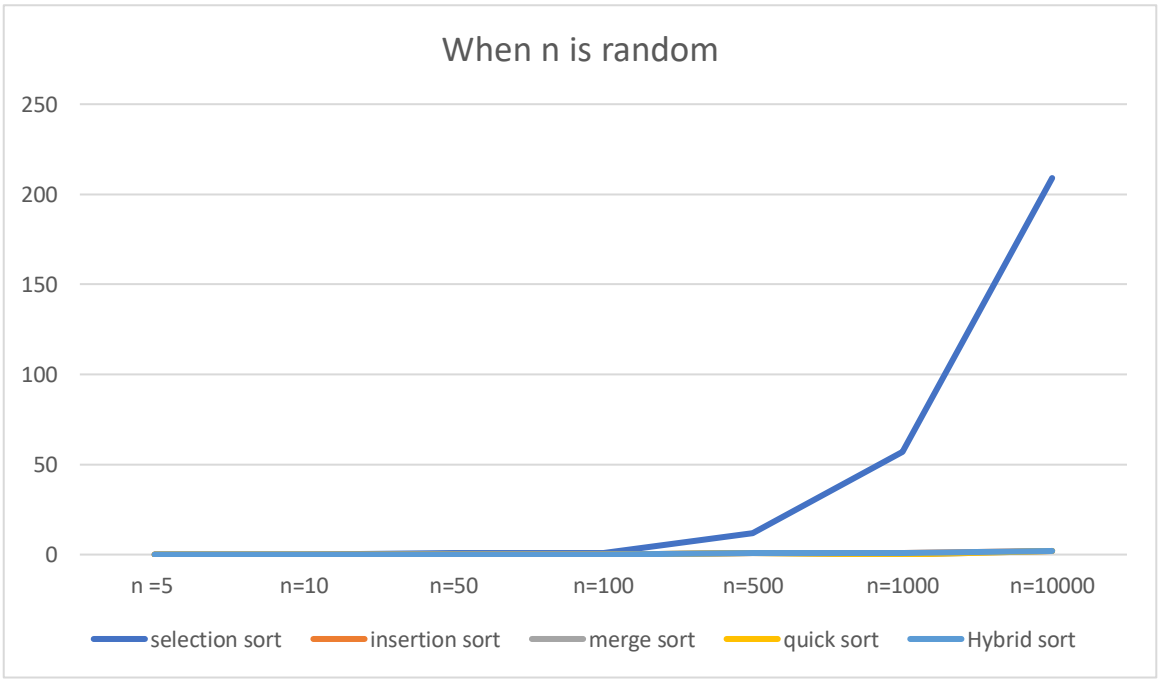
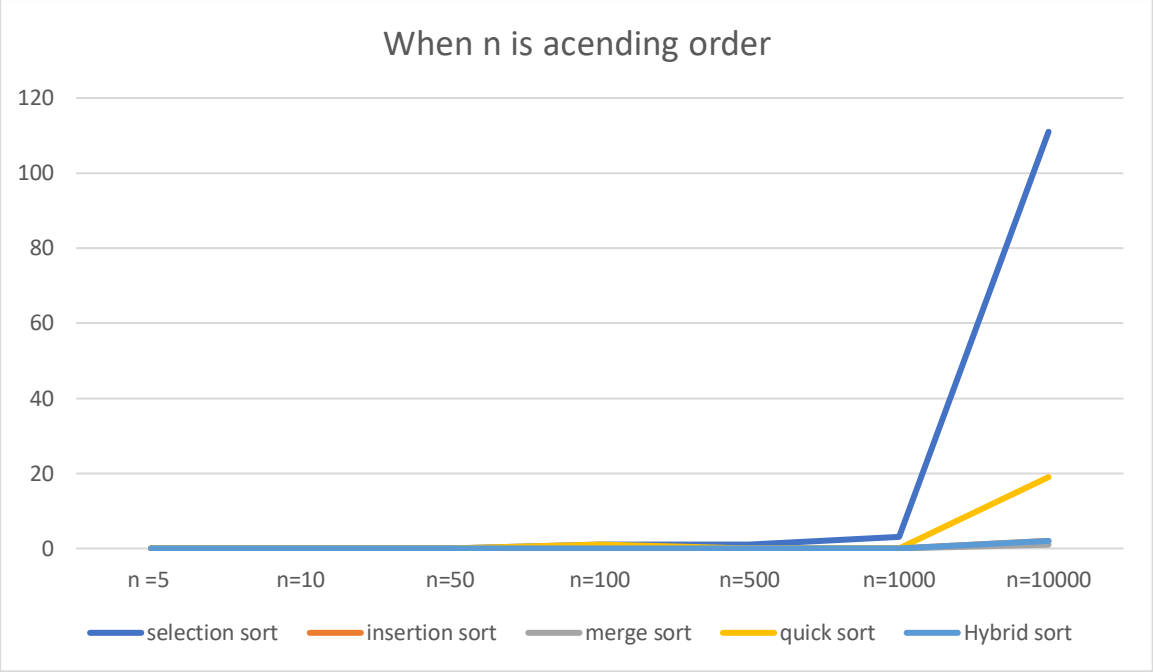
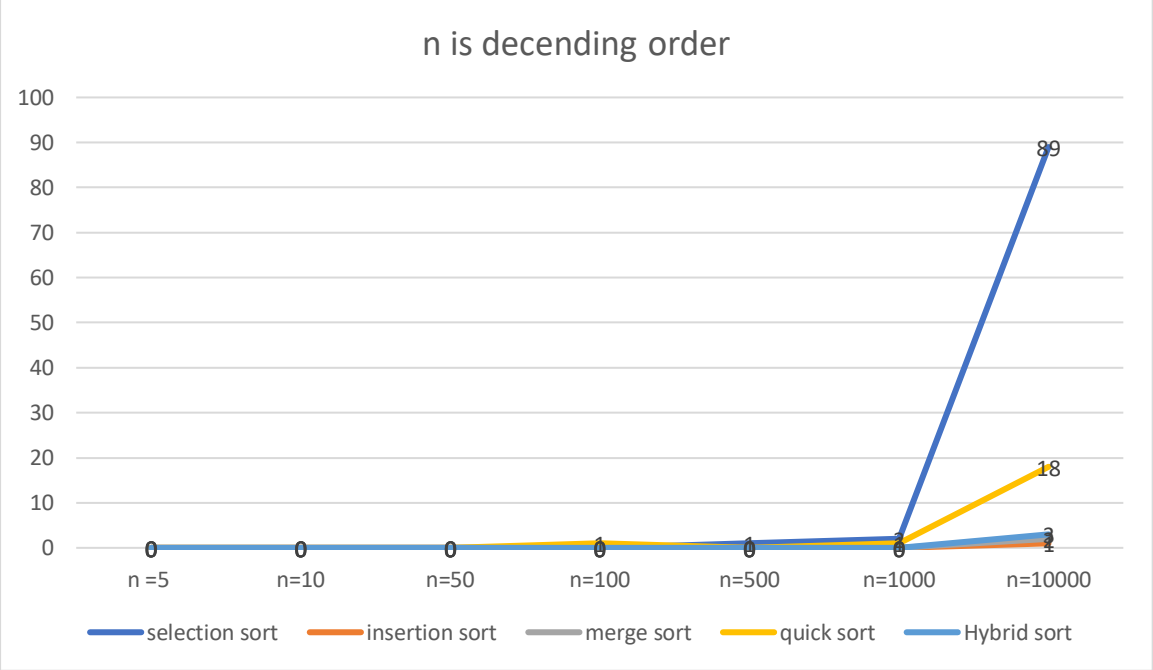
5.

Our observations on the sorting algorithms has lead to a conclusion that the asymptotically some sorting algorithm are better like merge sort with nlogn is better than selection and insertion sort but surprisingly inserting sort is faster than the merge sort until the n has grown to a considerable size. Also, quick sort does better when the array is random, but insertion sort is doing great work when the array is sorted and when n is less than or equal to 500. Coming to the quick sort, the efficiency depends on the how the pivoted element is been chosen for the list. Selection sort looks like not all good as its runtime is huge when n is great than 10 and when n is less than 10, selection sort is as good as selection sort and selection sort offers more incentives like when array is sorted, its runtime O(n). Coming to the merge sort, it si going better than quick sort when the array is sorted but selection sort will do even better and also merge sort has static run time nlogn but quick sort is starting to do good when the n value is raising like after 500. So, we can conclude that for smaller and sorted arrays, insertion sort and for larger and random array we use quick sort. Also, in quick let's use a better algorithm to find pivot element to find an optimal pivot on which runtime of quick sort depends a lot.

So, we are about to make a simple combination of the insertion and quick sort. While we always go for insertion sort of the array is sorted and quick sort if the array larger than 1000 elements and the array is not sorted. Coming to the quick sort, we observed that the random pivot will solve the orchestration of worst case and reputation of the worst case is not possible rather than picking the mid value as pivot.







Here, we can observe that the performance of quick sort improved.

Code

```
import java.util.Random;

public class HybridBestSortingAlgorithm {

	static <T extends Comparable> void hybridSort(T[] input,Boolean reversed)
{
//		System.out.println("HI ");

//		Integer input[] = new Integer[4];
//		input[0] = 2;
//		input[1] = 4;
//		input[2] = 3;
//		input[3] = 1;
//		sa.selectionSort(input, false);


		if (input.length <= 1000) {
			insertionSort(input, reversed);
		} else if (isArrySortedBinarySearch(input)) {
			insertionSort(input, reversed);

		} else {
			quickSortForProperInputs(input, 0, input.length-1, reversed);
		}
//		for(int i=0; i<input.length;i++) {
//			System.out.println(input[i]);
//		}
	}


	static <T extends Comparable> void insertionSort(T[] input, boolean
reversed) {

		int n = input.length;

		for (int i = 1; i < n; ++i) {
			T currentValue = input[i];
			int j = i - 1;

			/*
			 * Move elements of input from index 0 to i-1, which are
greater than
			 * currentValue to the one position ahead of their current
position
			 */
			while (j >= 0 && input[j].compareTo(currentValue) > 0 &&
reversed == false) {
				input[j + 1] = input[j];
				j = j - 1;
```

```
                }
                /*
                 * Move elements of input from index 0 to i-1, which are
lesser than
                 * currentValue to the one position ahead of their current
position
                 */
                while (j >= 0 && input[j].compareTo(currentValue) < 0 &&
reversed == true) {
                        input[j + 1] = input[j];
                        j = j - 1;
                }
                input[j + 1] = currentValue;
        }

    }

    static <T extends Comparable> Boolean isArrySortedBinarySearch(T arr[]) {
        int l = 0, r = arr.length - 1;
        T x = arr[r];
        int counter = 0;
        int result = (int) (Math.log(r) / Math.log(2));
        result = result - 2;
        while (l <= r) {
            int m = l + (r - l) / 2;

            // Check if x is present at mid

            // If x greater, ignore left half
            if (arr[m].compareTo(x) < 0) {
                l = m + 1;
                counter++;
            }

            // If x is smaller, ignore right half
            else {
                if (counter >= result)
                    return true;
                else
                    return false;

            }

        }

        if (counter >= result)
            return true;
        else
            return false;

    }

    private static <T extends Comparable> void quickSortForProperInputs(T[]
input, int start, int end,
            boolean reversed) {
        // TODO Auto-generated method stub
        if (start < end) {
            int dividingIndex = quickSortinActionForPartion(input, start,
end, reversed);
```

```
                    // here we are dividing the array in to two halfs, form start
of the array to
                    // mid
                    quickSortForProperInputs(input, start, dividingIndex - 1,
reversed);
                    // here we are dividing the array in to two halfs, form mid of
the array to last
                    quickSortForProperInputs(input, dividingIndex + 1, end,
reversed);
             }
      }

      private static <T extends Comparable> int quickSortinActionForPartion(T[]
input, int start, int end,
                    boolean reversed) {
             // TODO Auto-generated method stub
             // Picking the median of first, last and mid element.

             int length = end - start + 1;
             Random Dice = new Random();
             int n = Dice.nextInt(length);
             T pivot = input[n];
             int i = (start - 1);

             for (int j = start; j < end; j++) {

                    if (input[j].compareTo(pivot) <= 0 && reversed == false) {
                           i++;

                           T TemperaryVariableToRepace = input[i];
                           input[i] = input[j];
                           input[j] = TemperaryVariableToRepace;
                    } else if (input[j].compareTo(pivot) >= 0 && reversed == true)
{
                           i++;

                           T TemperaryVariableToRepace = input[i];
                           input[i] = input[j];
                           input[j] = TemperaryVariableToRepace;
                    }
             }

             T TemperaryVariableToRepace = input[i + 1];
             input[i + 1] = input[end];
             input[end] = TemperaryVariableToRepace;

             return i + 1;
      }

}
```