INTRODUCTION

In this programming assignment, I designed an experimental study for the comparison between Selection Sort, Insertion Sort, Merge Sort, Quick Sort with Lomuto & Hoare partitioning, and Heapsort algorithms. I designed the experiments for the comparison of two algorithms both theoretically and empirically.

QUICKSORT ALGORITHM

I implemented parts of Hoare' and Lomuto's partitioning in Java programming language. Implementations are in Java source code file.

```java
public static int LomutoPartition(int[] b, int low, int high) {
    int pivot = b[high];
    int i = low;
    for (int j = low; j < high; j++) {
        if (b[j] <= pivot) {
            swap(b, i, j);
            i++;
        }
    }
    swap(b, i, high);

    return i;
}

public static void LomutosQuickSort(int[] b, int low, int high) {

    if (low < high) {
        int p = LomutoPartition(b, low, high);
        LomutosQuickSort(b, 0, p - 1);
        LomutosQuickSort(b, p + 1, high);
    }

}
```

```java
public static int HoarePartition(int[] a, int low, int high) {
    int pivot = a[low];
    int i = low - 1;
    int j = high + 1;

    while (true) {
        do {
            i++;
        } while (a[i] < pivot);

        do {
            j--;
        } while (a[j] > pivot);

        if (i >= j) {
            return j;
        }

        swap(a, i, j);
    }
}

public static void quicksortHoares(int[] a, int low, int high) {
    if (low >= high) {
        return;
    }
    int pivot = HoarePartition(a, low, high);
    //elements less than the pivot
    quicksortHoares(a, low, pivot);
    //elements more than the pivot
    quicksortHoares(a, pivot + 1, high);
}
```

The pseudocode of the codes is as follows:

- **Pseudocode of Hoare's partitioning:**

HoarePartition (a[], low, high)

pivot = a[low]

i = low - 1

j = high + 1

repat i = i + 1 until a[i] < pivot

repat j-- until (a[j] > pivot);

if i >= j then

return j

```
 swap a[i] with a[j]
quicksortHoares(int[] a, int low, int high)
 if low >= high then
 return;
 pivot = HoarePartition(a, low, high)
 //elements less than the pivot
 quicksortHoares(a, low, pivot)
 //elements more than the pivot
 quicksortHoares(a, pivot + 1, high)
```

- **Pseudocode of Lomuto's partitioning:**

```
LomutoPartition (b[], low, high)
 pivot = b[high];
 i =low;
 for j=low; j<high; j++
 if b[j] <= pivot
swap b[i] with b[j]
 i++;
 swap b[i] with b[high]
 return i;
LomutosQuickSort(int[] b, int low, int high)

 if low < high then
 p = LomutoPartition(b, low, high)
 LomutosQuickSort(b, 0, p - 1)
 LomutosQuickSort(b, p + 1, high)
```

Time Complexity of Hoare's and Lomuto's Partitions

The runtime of quicksort depends on whether the shredding is balanced or unbalanced. If partitioning is balanced, the algorithm runs fast by asymptotically sorting. However, if partitioning is unbalanced, it runs asymptotically as slow as insertion sorting algorithm.

- Lomuto's Partition Time Complexity:

Basic Operation: Comparison

Input Size: n

Time Complexity: n+5 ∈ O(n)

- LomutosQuickSort method's time complexity:

Best and Worst Case:

A[p…r-1] > pivot

T(n) = T(n-1)+n   T(n) ∈ O(n*n)

INSERTIONSORT

Algorithm for Insertion Sort:

Step 1 – If the element is the first one, it is already sorted.
Step 2 – Move to next element
Step 3 – Compare the current element with all elements in the sorted array
Step 4 – If the element in the sorted array is smaller than the current element, iterate to the next element. Otherwise, shift all the greater element in the array by one position towards the right
Step 5 – Insert the value at the correct position
Step 6 – Repeat until the complete list is sorted

Implementation:

```
/**
 *
 * @author ercel
 */
import java.util.Arrays;
public class InsertionSort
{
    void insertionSort(int array[]) {
    for (int i = 1; i < array.length; i++) {
      int deger = array[i];
      int j = i - 1;
      while (j >= 0 && deger < array[j]) {
        array[j + 1] = array[j];
        --j;
      }
      array[j + 1] = deger;
    }
  }

   public static void main(String args[]) {
    int[] sayilar = { 9, 5, 1, 4, 3 };
    InsertionSort _insertSort = new InsertionSort();
    _insertSort.insertionSort(sayilar);
    System.out.println("Sorted Array in Ascending Order: ");
    System.out.println(Arrays.toString(sayilar));
  }
}
```

Psedeucode of insertionsort:

```
INSERTION-SORT(A)
for j ← 2 to length[A]
  do key ← A[j]
    ▹ Insert A[j] into the sorted sequence A[1  j - 1].
   i ← j - 1
   while i > 0 and A[i] > key
    do A[i + 1] ← A[i]
       i ← i - 1
   A[i + 1] ← key
```

Time complexity:

Time Complexity: O(N^2)   ->   array is reversely sorted
Average case: O(N^2)   ->   array is randomly sorted
Best case complexity: O(n)   ->   array is already sorted
Auxiliary Space: O(1)   because an extra variable key is used.
Stability: Yes
Inplace: Yes

SELECTIONSORT

Algorithm of SelectionSort:

Step 1: Set Min to location 0 in Step 1.
Step 2: Look for the smallest element on the list.
Step 3: Replace the value at location Min with a different value.
Step 4: Increase Min to point to the next element
Step 5: Continue until the list is sorted.

```java
public class SelectionSort
{
    void selectionSort(int array[]) {

    for (int i = 0; i < array.length - 1; i++) {
      int _minumumSayi = i;
      for (int j = i + 1; j < array.length; j++) {

        // To sort in descending order, change > to < in this line.
        // Select the minimum element in each loop.
        if (array[j] < array[_minumumSayi]) {
          _minumumSayi = j;
        }
      }

      // put min at the correct position
      int geciciDeger = array[i];
      array[i] = array[_minumumSayi];
      array[_minumumSayi] = geciciDeger;
    }
  }
    public static void main(String[] args) {
        int[] sayilar = { 20, 12, 10, 15, 2 };
        SelectionSort _selectSort = new SelectionSort();
        _selectSort.selectionSort(sayilar);
        System.out.println("Sorted Array in Ascending Order: ");
        System.out.println(Arrays.toString(sayilar));
    }
}
```

The selection sort pseudocode is as follows:

array : array of items
size : size of list
for i = 1 to size - 1
minimum = i // set current element as minimum
for j = i+1 to n // check the element to be minimum
if array[j] < array[minimum] then
minimum = j;
end if
end for
if indexofMinimum != i then //swap the minimum element with the current element
swap array[minimum] and array[i]
end if
end for
end function

Time Complexity:

Time Complexity: O(n^2)

Average Case: $O(n^2)$
Best case complexity: $O(n^2)$
Auxiliary Space: O(1)

Selection Sort Applications:

- The selection sort is used when
- a small list is to be sorted
- cost of swapping does not matter
- checking of all the elements is compulsory
- cost of writing to a memory matters like in flash memory (number of writes/swaps is O(n) as compared to $O(n^2)$ of bubble sort)

MERGESORT

➔ Merge sort algorithm is an example of a divide and conquer sorting algorithm. The algorithm is a recursive one which divides a list of data items into halves, sorts each half of data items separately and then merges both halves into one sorted array.

MergeSort Algortihm Steps:

Step 1: Find the middle index of the array.
Middle = 1 + (last – first)/2
Step 2: Divide the array from the middle.
Step 3: Call merge sort for the first half of the array
MergeSort(array, first, middle)
Step 4: Call merge sort for the second half of the array.
MergeSort(array, middle+1, last)
Step 5: Merge the two sorted halves into a single sorted array.

Implementation:

```
void merge(int arr[], int p, int q, int r) { // Create L ← A[p..q] and M ← A[q+1..r]
    int n1 = q - p + 1;
    int n2 = r - q;

    int L[] = new int[n1];
    int M[] = new int[n2];

    for (int i = 0; i < n1; i++) {
        L[i] = arr[p + i];
    }
    for (int j = 0; j < n2; j++) {
        M[j] = arr[q + 1 + j];
    }
// Maintain current index of sub-arrays and main array
    int i, j, k;
    i = 0;
    j = 0;
    k = p;

    // Until we reach either end of either L or M, pick larger among
    // elements L and M and place them in the correct position at A[p..r]
    while (i < n1 && j < n2) {
        if (L[i] <= M[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = M[j];
            j++;
        }
        k++;
```

```
    while (i < n1 && j < n2) {
        if (L[i] <= M[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = M[j];
            j++;
        }
        k++;
    }

    // When we run out of elements in either L or M,
    // pick up the remaining elements and put in A[p..r]
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = M[j];
        j++;
        k++;
    }
}
```

```java
    void mergeSort(int arr[], int l, int r) {
        if (l < r) {

            // m is the point where the array is divided into two subarrays
            int m = (l + r) / 2;

            mergeSort(arr, l, m);
            mergeSort(arr, m + 1, r);

            // Merge the sorted subarrays
            merge(arr, l, m, r);
        }
    }
    // Print the array
    static void printArray(int arr[]) {
        int n = arr.length;
        for (int i = 0; i < n; ++i) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }     // Driver program
    public static void main(String args[]) {
        int sayilar[] = {6, 5, 12, 10, 9, 1};

        MergeSort _mergeSort = new MergeSort();
        _mergeSort.mergeSort(sayilar, 0, sayilar.length - 1);
        System.out.println("Sorted array:");
        printArray(sayilar);

    }
}
```

Pseudocode of MergeSort:

procedure merge(Arr[], lt, mid, rt):
int L1 = mid - lt + 1

int L2 = rt-mid

int left[L1], right[L2]

for i = 0 to L1:

left[i] = Arr[lt + i]

END for loop

for j = 0 to L2:

right[j] = Arr[mid+1+j]

END for loop

while(left and right hve elments):

if(left[i] < right[j])

Add left[i] to the end of Arr

else

Add right[i] to the end of Arr

END while loop

END procedure

procedure Merge_sort(Arr[]):

l1 = Merge_sort(L1)

l2 = Merge_sort(L2)

return merge(l1, l2)

END procedure

Time Complexity:

Worst case: O(n logn)

Best case: O(n logn)

Average Case: O(n logn)

Space Complexity: O(n)

Merge Sort Applications

Inversion count problem

External sorting

E-commerce applications

HEAPSORT

Heap Sort Algorithm
Step 1 - Construct a Binary Tree with given list of Elements.
Step 2 - Transform the Binary Tree into Min Heap.
Step 3 - Delete the root element from Min Heap using Heapify method.
Step 4 - Put the deleted element into the Sorted list.
Step 5 - Repeat the same until Min Heap becomes empty.

```java
class HeapSort {
  public void sort(int arr[]) {
    int n = arr.length;

    // Build max heap
    for (int i = n / 2 - 1; i >= 0; i--) {
      heapify(arr, n, i);
    }

    // Heap sort
    for (int i = n - 1; i >= 0; i--) {
      int temp = arr[0];
      arr[0] = arr[i];
      arr[i] = temp;

      // Heapify root element
      heapify(arr, i, 0);
    }
  }

  void heapify(int arr[], int n, int i) {
    // Find largest among root, left child and right child
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    if (l < n && arr[l] > arr[largest])
      largest = l;

    if (r < n && arr[r] > arr[largest])
      largest = r;

    // Swap and continue heapifying if root is not largest
    if (largest != i) {
      int swap = arr[i];
      arr[i] = arr[largest];
      arr[largest] = swap;

      heapify(arr, n, largest);
    }
  }

  // Function to print an array
  static void printArray(int arr[]) {
    int n = arr.length;
    for (int i = 0; i < n; ++i)
      System.out.print(arr[i] + " ");
    System.out.println();
  }
```

```java
    // Function to print an array
    static void printArray(int arr[]) {
      int n = arr.length;
      for (int i = 0; i < n; ++i)
        System.out.print(arr[i] + " ");
      System.out.println();
    }

    // Driver code
    public static void main(String args[]) {
      int sayilar[] = { 1, 12, 9, 5, 6, 10 };

      HeapSort _heapSort = new HeapSort();
      _heapSort.sort(sayilar);

      System.out.println("Sorted array is");
      printArray(sayilar);
    }
}
```

PsedeuCode of Heapsort:

```
   Heapify(A as array, n as int, i as int)
{
   max = i
   leftchild = 2i + 1
   rightchild = 2i + 2
   if (leftchild <= n) and (A[i] < A[leftchild])
      max = leftchild
   else
      max = i
   if (rightchild <= n) and (A[max]  > A[rightchild])
      max = rightchild
   if (max != i)
      swap(A[i], A[max])
      Heapify(A, n, max)
}
Heapsort(A as array)
{
  n = length(A)
  for i = n/2 downto 1
    Heapify(A, n ,i)

  for i = n downto 2
    exchange A[1] with A[i]
    A.heapsize = A.heapsize - 1
    Heapify(A, i, 0)
}
```

Time Complexity:

Worst case: O(nlog n)

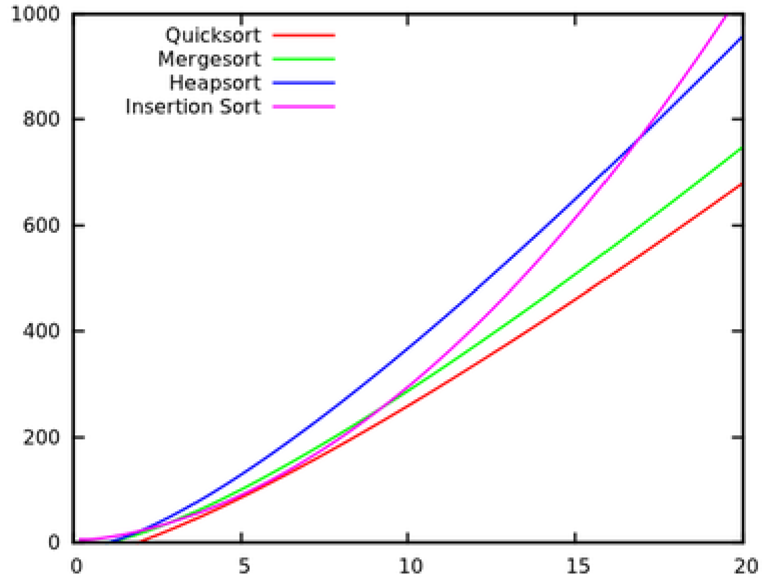
Best Case: O(nlog n)

Average Case: O(nlog n)

Space Complexity: O(1)

CPU TIME VALUES OF EACH ALGORITHMS

ALGORITHM	10	100	1k	10k	Best Case	Worst Case
InsertionSort	0.0818	6.831	757.851	77713.30	2.351	1615.53
SelectionSort	0.0690	6.453	507.285	46800.13	493.901	479.04
MergeSort	0.0964	2.836	34.649	491.92	16.687	20.o6
HeapSort	0.1986	4.872	67.710	1887.41	70.570	72.84
QuickSort	0.1115	2.211	26.759	907.60	96.938	691.21



RESULT AND DISCUSSION

The results indicate that, Lomuto's partitioning does the work in just one array traversal like as Hoare's partition, but Lomuto partition requires more swaps. Lomuto's partition puts the pivot at the correct position in the array as well as returns the index whereas Hoare's partition only returns the correct index of the pivot. Lomuto partition is more relatively inefficient in time complexity respect. This situation to cause Lomuto's partition slower than Hoare's partition. Both partitions are linear algorithms. Upon exploring the situation from multiple perspectives, we can say that, while Hoare's partition algorithm is slightly difficult to understand and to implement, Lomuto's partition algorithm easier to understand and implement. based on the results of this study, it seems some factors causes the Hoare's partitioning algorithm to be preferred. In conclusion, each sorting algorithm has its particular strengths and weaknesses. At times any sorting algorithm will work for variety of choices. However, most algorithms are judged by their time efficiency (best case, worst case and average case), space requirement and memory for a particular set of inputs. For instance, the quick sort algorithm performs well for some inputs but horrible for others hence one should consider such factors before deciding which algorithm to employ for sorting.