

Lab 4

Student

ID	Name
614760	Omkar Nath Chaudhary
614732	Sushil Subedi
614604	Rahul Niraula
614600	Shrawan Adhikari

Q1.

Insertion Sort:

E.g Given: $8_1, 8_2, 2, 1$

$8_1, 8_2, 2, 1$

$2, 8_1, 8_2, 1$

$1, 2, 8_1, 8_2$

Therefore, we get, $1, 2, 8_1, 8_2$ **Stable**

Bubble Sort:

E.g Given: $8_1, 8_2, 2, 1$

$8_2, 2, 1, 8_1$

$1, 2, 8_2, 8_1$

Therefore, we get, $1, 2, 8_2, 8_1$ **Not Stable**

Selection Sort: Not Stable

E.g Given: $8_1, 8_2, 2, 1$

Solving this using insertion sort:

$1, 8_2, 2, 8_1$

$1, 2, 8_2, 8_1$

Therefore, we get, $1, 2, 8_2, 8_1$ **Not Stable**

Insertion Sort is always Stable, whereas Bubble Sort and Selection Sort are not.

Q2.

Divide

7, 6, 5, 4, 3, 2, 1

7, 6, 5 4, 3, 2, 1

7 6, 5 4, 3 2, 1

7 6 5 4 3 2 1

Merge

7 5, 6 3, 4 1, 2

5, 6, 7 1, 2, 3, 4

1, 2, 3, 4, 5, 6, 7

Q3.

```
private static final int INSERTION_THRESHOLD = 20;
private int[] resArr;
```

```
private void merge(int[] tempStorage, int lower, int upperPointer, int upper) {
    int j = 0;
    int lowerBound = lower;
    int n = upper - lowerBound + 1;

    int mid = upperPointer - 1;

    while (lower <= mid && upperPointer <= upper) {
        if (resArr[lower] < resArr[upperPointer]) {
            tempStorage[j++] = resArr[lower++];
        } else {
            tempStorage[j++] = resArr[upperPointer++];
        }
    }
    while (lower <= mid) {
        tempStorage[j++] = resArr[lower++];
    }
}
```

```

while (upperPointer <= upper) {
    tempStorage[j++] = resArr[upperPointer++];
}

for (j = 0; j < n; ++j) {
    resArr[lowerBound + j] = tempStorage[j];
}

}

void mergeSort(int[] tempStorage, int lower, int upper) {
    if (lower == upper) {
        return;
    }
    int itemsCount = upper - lower;
    if (itemsCount <= this.INSERTION_THRESHOLD) {
        insertionSort(lower, upper);
    } else {
        int mid = (lower + upper) / 2;
        mergeSort(tempStorage, lower, mid);
        mergeSort(tempStorage, mid + 1, upper);
        merge(tempStorage, lower, mid + 1, upper);
    }
}

private void insertionSort(int lower, int upper) {
    if (resArr == null || resArr.length <= 1)
        return;

    int temp = 0;
    int j = 0;
    for (int i = lower; i <= upper; ++i) {
        temp = resArr[i];
        j = i;
        while (j > lower && temp < resArr[j - 1]) {
            resArr[j] = resArr[j - 1];
            j--;
        }
        resArr[j] = temp;
    }
}

public static void main(String[] args) {
    int[] input = {5, 4, 3, 7, 8, 1, 2};
    System.out.println("Input: " + Arrays.toString(input));

    MergeSort msp = new MergeSort();

```

```

int[] tempStorage = new int[input.length];
msp.resArr = input;
msp.mergeSort(tempStorage, 0, input.length - 1);
System.out.println("Result " + Arrays.toString(input));
}

```

Q4.

Handwritten diagrams illustrating binary trees and their heights:

- i)** A binary tree with height 2. It has a root node with two children. The left child has two children of its own, and the right child has one child. $\text{height} = 2$
- ii)** A binary tree with height 3. It has a root node with two children. The left child has two children, and the right child has two children. The rightmost child of the right child has two children. $\text{height} = 3$
- iii)** A binary tree with height 3. It has a root node with two children. The left child has two children, and the right child has two children. The rightmost child of the right child has two children. $\text{height} = 3$
- iv)** A binary tree with height 3. It has a root node with two children. The left child has two children, and the right child has two children. The rightmost child of the right child has two children. $\text{height} = 3$

Handwritten text below the diagrams:

b) Every binary tree of height 3 has at most $2^3 = 8$ leaves which is true from above proof solution.

c) The number of leaves of a binary tree of height n has at most 2^n leaves.