Recursive Functions

```
public static int fib(int n) {
    if (n < 2) {
        return 1;
    }

    return fib(n-1) + fib(n-2);
}
```

```
public static void printReverse(int[] values, int idx) {
    if (idx > 0) {
        System.out.print(values[idx] + ", ");
        printReverse(values, idx - 1);
    }
}
```

Insertion Sort

```
public static void swap(int[] values, int i, int j) {
    int tmp = values[i];
    values[i] = values[j];
    values[j] = tmp;
}

public static void insertionSort(int[] values) {
    int i = 1;
    while (i <  values.length) {
        int j = i;
        while (j > 0 && values[j-1] > values[j]) {
            swap(values, j, j-1);
            j--;
        }
        i++;
    }
}
```

Mergesort

```
public static int[] merge(int[] left, int[] right) {
    int[] result = new int[left.length + right.length];
    int leftIndex = 0;
    int rightIndex = 0;
    int resultIndex = 0;

    while (leftIndex < left.length && rightIndex < right.length) {
        if (left[leftIndex] <= right[rightIndex]) {
            result[resultIndex] = left[leftIndex];
            leftIndex++;
            resultIndex++;
        } else {
            result[resultIndex] = right[rightIndex];
            rightIndex++;
            resultIndex++;
        }
    }

    while (leftIndex < left.length) {
        result[resultIndex] = left[leftIndex];
        leftIndex++;
        resultIndex++;
    }

    while (rightIndex < right.length) {
        result[resultIndex] = right[rightIndex];
        rightIndex++;
        resultIndex++;
    }

    return result;
}

public static int[] mergesort(int[] values) {
    if (values.length == 1) {
        return values;
    }

    int leftSize = values.length / 2;
```

```
    int rightSize = values.length - leftSize;

    int[] left = new int[leftSize];
    int leftIndex = 0;

    int[] right = new int[rightSize];
    int rightIndex = 0;

    for (int i = 0; i < values.length; i++) {
        if (i < values.length/2) {
            left[leftIndex] = values[i];
            leftIndex++;
        } else {
            right[rightIndex] = values[i];
            rightIndex++;
        }
    }

    left = mergesort(left);
    right = mergesort(right);

    return merge(left, right);
}
```

Quicksort

```
public static void swap(int[] values, int i, int j) {
    int tmp = values[i];
    values[i] = values[j];
    values[j] = tmp;
}

public static void quicksort(int[] values, int lo, int hi) {
    if (lo < hi) {
        int p = partition(values, lo, hi);
        quicksort(values, lo, p-1);
        quicksort(values, p+1, hi);
    }
}

public static int partition(int[] values, int lo, int hi) {
```

```
    int pivot = values[hi];
    int i = lo;

    for (int j = lo; j < hi; j++) {
        if (values[j] < pivot) {
            swap(values, i, j);
            i++;
        }
    }

    swap(values, i, hi);

    return i;
}
```