Search and Sort Data

Iterative

```
//TODO: Complete this search algorithm to find an element location is in array
//      use while loops here
public static int binarySearch(int[] data, int target){
    //TODO: update with search algorithm
    int left = 0;
    int right = data.length - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (data[mid] == target) {
            return mid;
        } else if (data[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1; // target not found
}
```

binarySearch(int[] data, int target): This method implements the binary search algorithm using a while loop. It starts by setting two pointers at the beginning and end of the array and calculates the middle index. It then compares the middle element to the target value and adjusts the pointers accordingly until the target is found or the pointers meet.

```
//TODO: Complete the following sorting algorithm
public static void bubbleSort(int[] data) {
    //TODO: Implement the algorithm for this sort.
    int n = data.length;
    boolean swapped;

    do {
        swapped = false;
        for (int i = 1; i < n; i++) {
            if (data[i - 1] > data[i]) {
                swap(data, i - 1, i);
                swapped = true;
            }
        }
        n--;
    } while (swapped);
}
```

bubbleSort(int[] data): The bubbleSort method implements the bubble sort algorithm using a do-while loop. It iterates through the array and compares adjacent elements, swapping them if they are in the wrong order. This process is repeated until no more swaps are needed, indicating

that the array is sorted.

```
//TODO: Complete this search algorithm to check if an element is in array
public static boolean contains(int[] data, int target) {
    //TODO: update with search algorithm
    for (int value : data) {
        if (value == target) {
            return true;
        }
    }
    return false;
}
```

contains(int[] data, int target): This method checks if a specific target element exists in the array by iterating through each element linearly. If the target is found, it returns true; otherwise, it returns false.

```
//TODO: Complete this search algorithm to find an element location is in array
//      use for loops here
public static int indexOf(int[] data, int target) {
    //TODO: update with search algorithm
    for (int i = 0; i < data.length; i++) {
        if (data[i] == target) {
            return i;
        }
    }
    return -1; // target not found
}
```

indexOf(int[] data, int target): The indexOf method searches for the index of a target element in the array by iterating through each element using a for loop. If the target is found, it returns the index; otherwise, it returns -1 to indicate that the target is not present in the array.

```
//TODO: Complete the following insertion sorting algorithm with the swap method
public static void insertionSort(int[] data) {
    //TODO: Implement the algorithm for this sort.
    for (int i = 1; i < data.length; i++) {
        int current = data[i];
        int j = i - 1;
        while (j >= 0 && data[j] > current) {
            data[j + 1] = data[j];
            j--;
        }
        data[j + 1] = current;
    }
}
```

insertionSort(int[] data): This method implements the insertion sort algorithm by iterating through the array and inserting each element into its correct sorted position within the already sorted part of the array.

```java
// This is an example of "finding the worst case run time function"
// for a maximum value algorithm
public static int max(int[] data){
    int n   = data.length;          // (1)      data size
    int max = data[0];              // (1)

    //       (1)
    for(int i = 0; i < n; i++)     // n times
        // (1  + 1)
        if(max < data[i])               // (1)
            max = data[i];              // (1)
    // (1)      terminates loop
    return max;                     // (1)

    // run time function f(n) = 1 + 1 + 1 + n(2 + 1 + 1)
    //                   f(n) = 4n + 5
}
```

max(int[] data): This method finds the maximum value in the array by iterating through the elements and updating the maximum value if a larger element is found. It returns the maximum value after the loop completes.

```java
//TODO: Complete the following selection sorting algorithm using swap method
public static void selectionSort(int[] data) {
    //TODO: Implement the algorithm for this sort.
    for (int i = 0; i < data.length - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < data.length; j++) {
            if (data[j] < data[minIndex]) {
                minIndex = j;
            }
        }
        swap(data, i, minIndex);
    }
}
```

selectionSort(int[] data): The selectionSort method implements the selection sort algorithm by repeatedly finding the minimum element from the unsorted part of the array and swapping it with the element at the current position.

```java
//TODO: Complete and use the following swap method for
//      sorting algorithm that require swapping of data
public static void swap(int[] data, int a, int b) {
    //TODO: Complete Body
    int temp = data[a];
    data[a] = data[b];
    data[b] = temp;
}
```

swap(int[] data, int a, int b): This method swaps two elements in an array given their indices a and b by using a temporary variable to hold one element's value while performing the swap.

Recursive

```
public static int binarySearch(int[] data, int target) {
    int low = 0;
    int high = data.length - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (data[mid] == target) {
            return mid;
        } else if (data[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    return -1; // Target not found
}
```

binarySearch(int[] data, int target): This method implements the binary search algorithm to find the index of the target element in the sorted array data. It uses an iterative approach to repeatedly divide the search space in half until the target is found or the search space is empty.

```
//TODO: Complete the following helper method for the corresponding
//      sorting algorithm
private static int[] getFirstHalf(int[] data) {
    //TODO : update to get first half of array
    int size = data.length / 2;
    int[] firstHalf = new int[size];
    System.arraycopy(data, 0, firstHalf, 0, size);
    return firstHalf;
}
```

getFirstHalf(int[] data): This helper method returns the first half of the input array data. It determines the size of the first half by dividing the length of the array by 2 and creates a new array with that size. It then copies the elements from the original array into the new array.

```java
//TODO: Complete the following helper method for the corresponding
//      sorting algorithm
private static int[] getSecondHalf(int[] data) {
    //TODO : update to get second half of array
    int size1 = data.length / 2;
    int size2 = data.length - size1;
    int[] secondHalf = new int[size2];
    System.arraycopy(data, size1, secondHalf, 0, size2);
    return secondHalf;
}
```

getSecondHalf(int[] data): This helper method returns the second half of the input array data. It determines the size of the second half by subtracting the size of the first half from the total length of the array. It creates a new array with that size and copies the elements from the original array starting from the index after the last element of the first half.

```java
//TODO: Complete the following helper method for the corresponding
//      sorting algorithm
private static void merge(int[] data, int[] left, int[] right) {
    //TODO: complete body
    int i = 0, j = 0, k = 0;
    while (i < left.length && j < right.length) {
        if (left[i] < right[j]) {
            data[k++] = left[i++];
        } else {
            data[k++] = right[j++];
        }
    }
    while (i < left.length) {
        data[k++] = left[i++];
    }
    while (j < right.length) {
        data[k++] = right[j++];
    }
}
```

merge(int[] data, int[] left, int[] right): This helper method merges two sorted arrays left and right into a single sorted array data. It uses three pointers (i, j, and k) to iterate through the elements of left, right, and data respectively. It compares the current elements at left[i] and right[j] and inserts the smaller element into data[k]. It continues this process until all elements from both left and right are merged into data.

```java
//TODO: Complete the following sorting algorithm
public static void mergeSort(int[] data) {
    //TODO: RECURSIVE CASE
    //TODO: complete body
    if (data.length > 1) {
        int[] left = getFirstHalf(data);
        int[] right = getSecondHalf(data);

        mergeSort(left);
        mergeSort(right);

        merge(data, left, right);
    }

}
```

mergeSort(int[] data): This method implements the merge sort algorithm to sort the array data in ascending order. It uses a recursive approach to divide the array into halves until each subarray contains only one element. Then, it recursively merges the sorted subarrays using the merge helper method to obtain the final sorted array.

```java
//TODO: Complete the following helper method for the corresponding
//      sorting algorithm
private static int partition(int[] data, int low, int high) {
    //TODO: update with partition algorithm
    int pivot = data[high];
    int i = (low - 1); // Index of smaller element

    for (int j = low; j < high; j++) {
        // If current element is smaller than or equal to pivot
        if (data[j] <= pivot) {
            i++;

            // Swap data[i] and data[j]
            int temp = data[i];
            data[i] = data[j];
            data[j] = temp;
        }
    }

    // Swap data[i+1] and data[high] (or pivot)
    int temp = data[i + 1];
    data[i + 1] = data[high];
    data[high] = temp;

    return i + 1; // Return the partitioning index
}
```

partition(int[] data, int low, int high): This helper method implements the partition step of the quicksort algorithm. It selects the last element data[high] as the pivot and rearranges the

elements such that all elements smaller than the pivot are placed before it, and all elements greater than the pivot are placed after it. It returns the index of the pivot element after partitioning.

```
//TODO: Complete the following sorting algorithm
public static void quickSort(int[] data) {
    //TODO : update to call helper method
    quickSort(data, 0, data.length - 1);
}
```

quickSort(int[] data): This method implements the quicksort algorithm to sort the array data in ascending order. It uses a recursive approach to divide the array into subarrays by selecting a pivot using the partition helper method. It then recursively sorts the left and right subarrays by calling itself on each subarray.

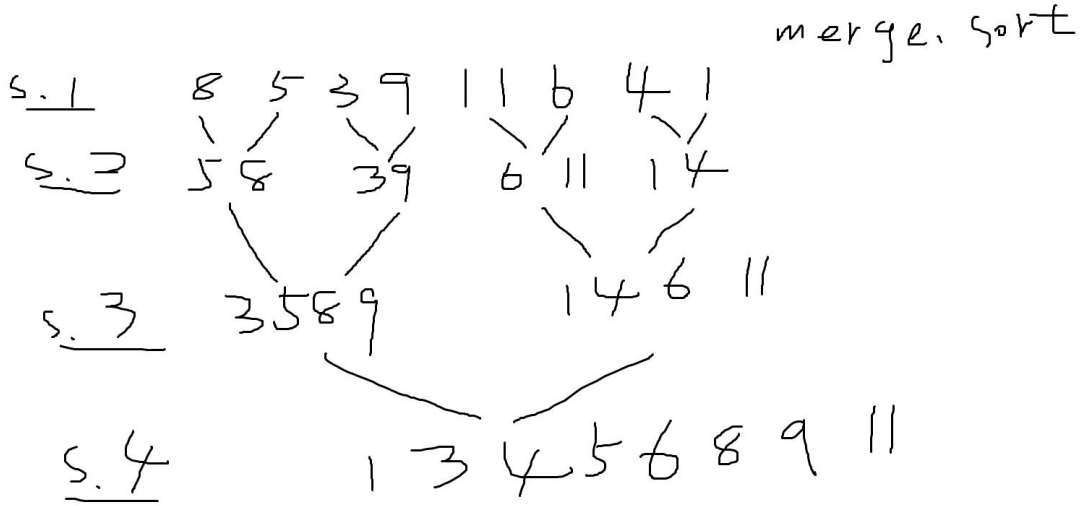
```
//TODO: Complete the following sorting algorithm
public static void selectionSort(int[] data) {
    //TODO: complete body
    selectionSort(data, 0);
}
```

selectionSort(int[] data): This method implements the selection sort algorithm to sort the array data in ascending order. It iteratively selects the minimum element from the unsorted portion of the array and swaps it with the first element of the unsorted portion. This process is repeated until the entire array is sorted.

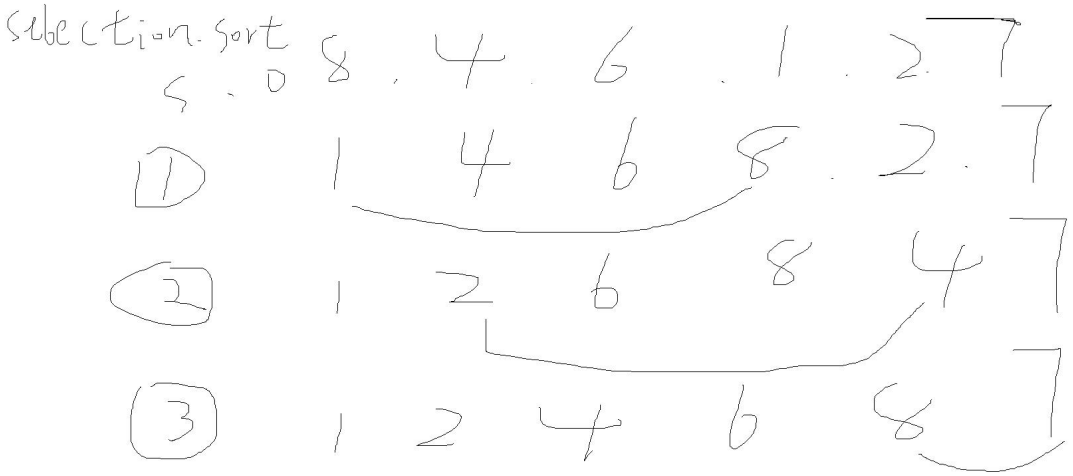
```
//TODO: Complete and use the following swap method for
//      sorting algorithm that require swapping of data
public static void swap(int[] data, int a, int b) {
    //TODO: complete body
    int temp = data[a];
    data[a] = data[b];
    data[b] = temp;
}
```

swap(int[] data, int a, int b): This helper method swaps the elements at indices a and b in the array data by using a temporary variable to store one of the elements temporarily.

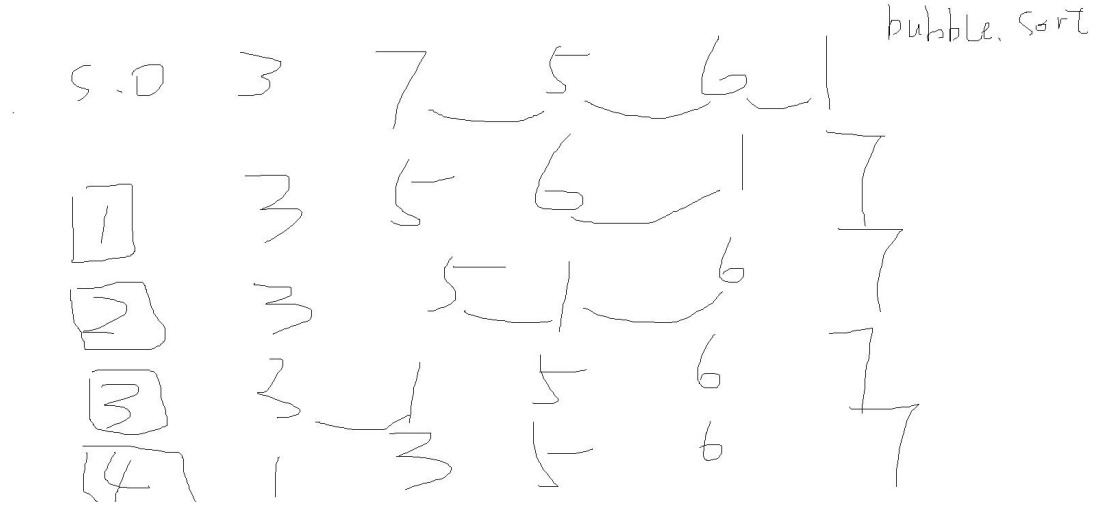
MergeSort



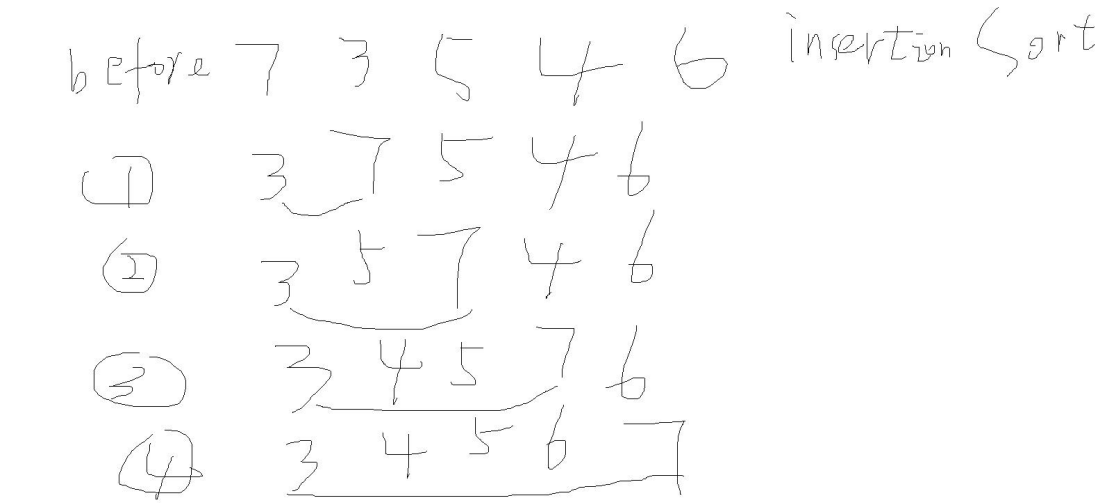
SelectionSort



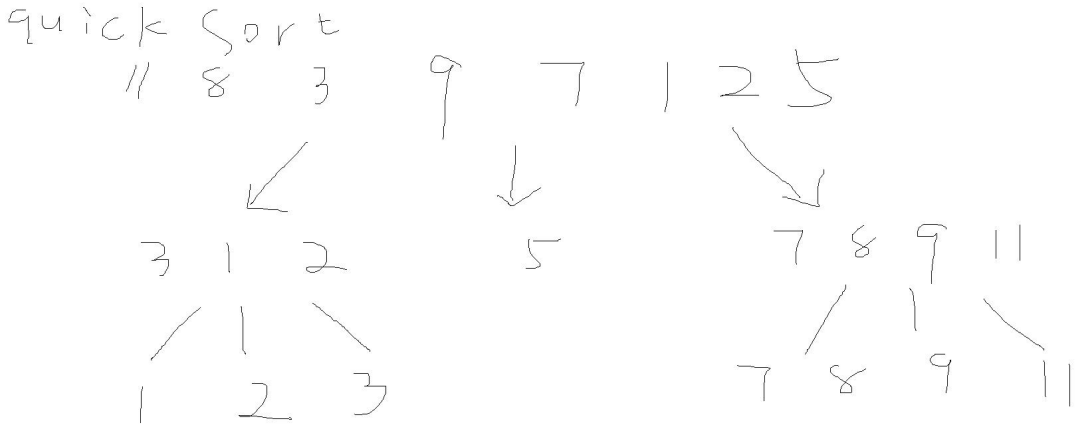
BubbleSort



InsertionSort



QuickSort



BinarySearch