Merge Sort

Merge sort is a type of sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the array into two halves until the subarrays only contain one element and cannot further be divided. Then the subarrays are merged in a sorted manner until the entire array is sorted.

Merge sort is a stable and consistent algorithm with a time complexity of O(n log(n)) in all scenarios. Its best-case performance happens when the array is already or nearly sorted and average-case is when the array is randomly sorted. The worst-case time complexity would be if the array is sorted in reversed order. When implemented for linked lists, the space complexity of the merge sort is O(log(n)).

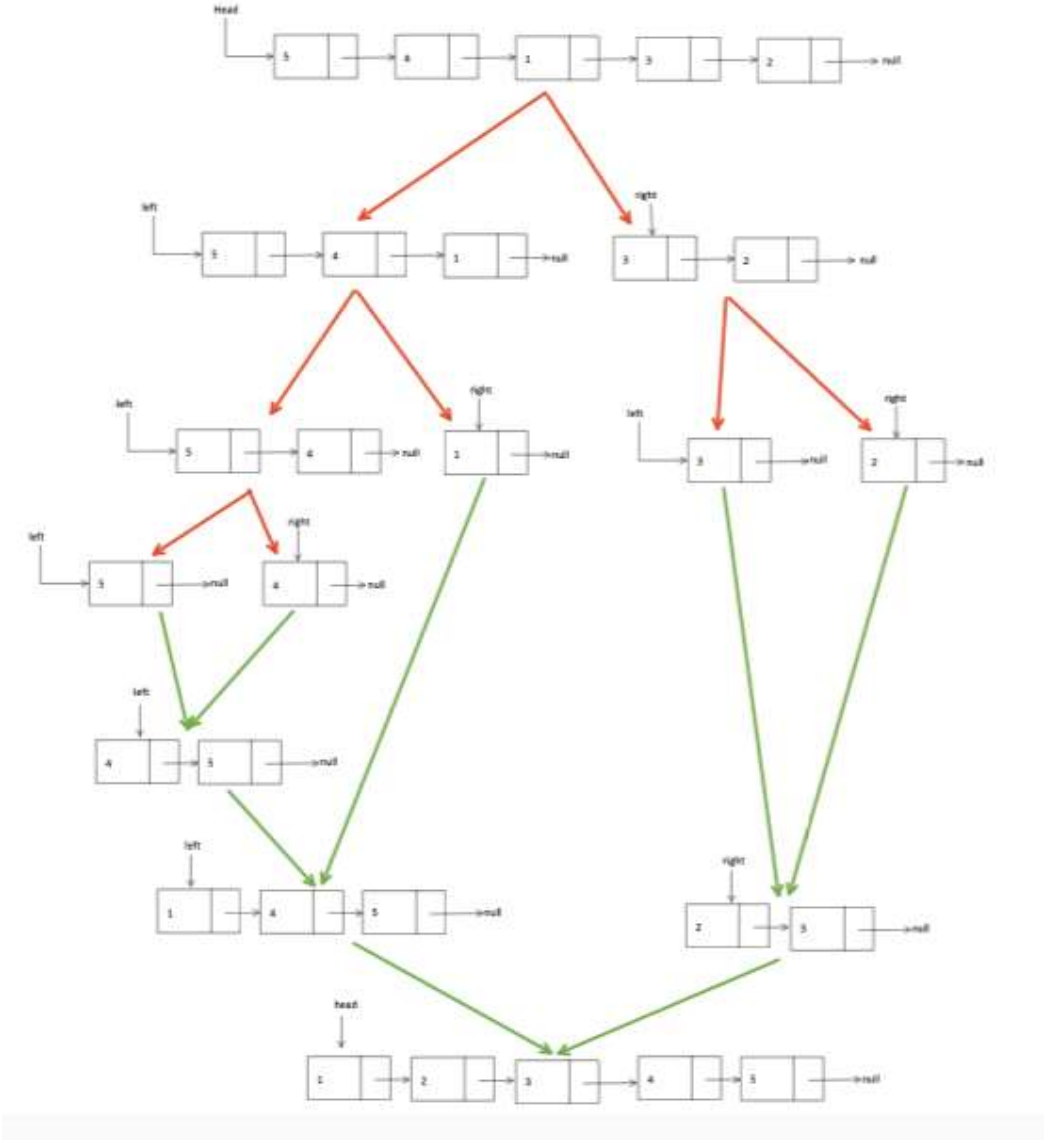
Merge sort is suitable for arrays as well as linked lists.

a)

We implemented merge sort with both ArrayList and Linked List and noticed that using linked lists over arrays is more efficient.
Linked lists splits into smaller sublists using pointers, whereas arrays require extra space memory to create new arrays during the splitting and merging processes.
In addition, the space complexity for merge sort implementation in arrays is O(n) while it's O(log n) in linked lists.
Below follows explanation for merge sort using both data structures:

Merge Sort implemented with linked list:

An overview of the method, with red arrows showing the splitting of the lists, and green arrows are the merges:



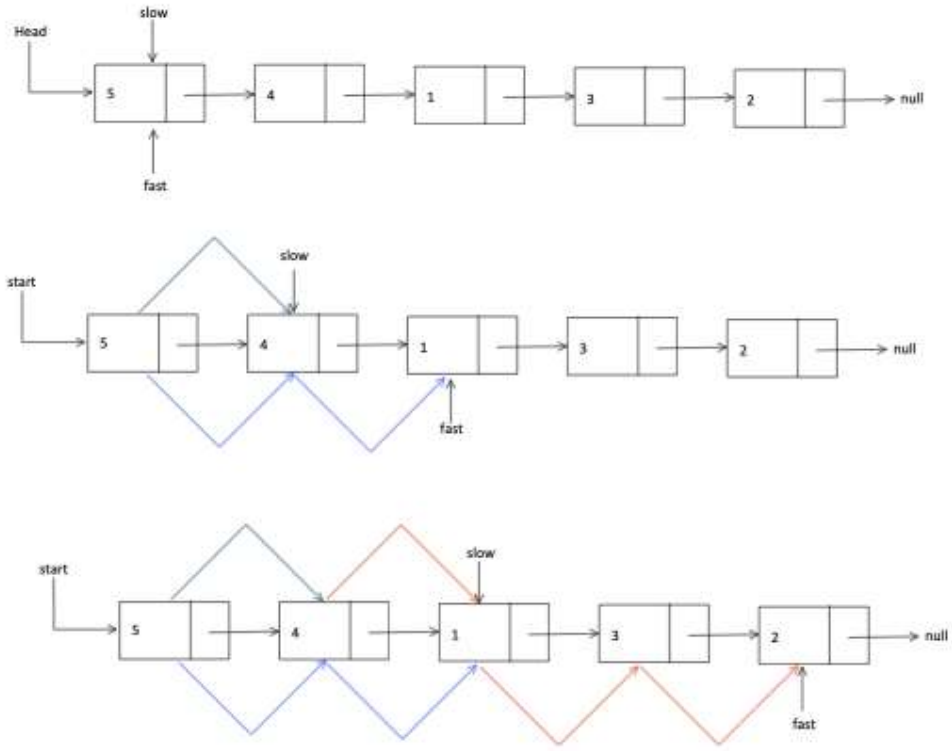
The linked list method takes in the head node of a linked list, referred to as ‘start’. We established a base case, to prevent infinite recursive looping, which stops dividing the list when a list contains only one node.

```
if (start == null || start.next == null) {
        return start;
    }
```

We determined the middle of the list by calling the ‘getMiddle’ method and passing in the start node. The ‘getMiddle’ method checks if the node is null; if not, it utilizes a pair of pointers - a slow and fast one - to traverse the list. The fast pointer moves twice as quickly as the slow one. If the fast pointer's next node, or its next's next

node is null, the loop terminates. Otherwise, the slow pointer increments once per loop, and fast pointer increments twice.
When the fast pointer reaches the end of the list or its node is null, the while-loop stops, and the slow pointer points to the middle node. The method then returns this node.

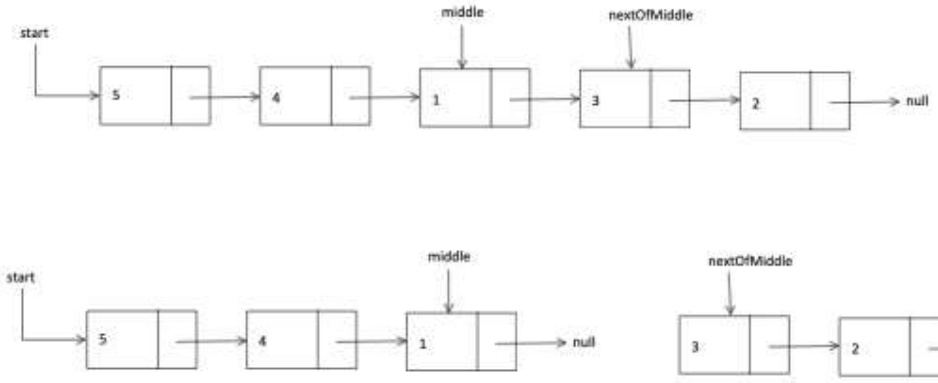
Here is a visual representation:




```
public static cityNode getMiddle(cityNode start) { //static
        if (start == null) {
            return start;
        }
        cityNode slow = start;
        cityNode fast = start;
        while ((fast.next != null) && (fast.next.next != null)) {
            slow = slow.next;
            fast = fast.next.next;
        }
        return slow;
    }
```

After finding the middle node, a new pointer, ‘nextOfMiddle’, is created, pointing to the node after the middle node. The middle's next is set to null, so that the list is split

into two separate lists: one from the start node to the middle node, and another from ‘nextOfMiddle’ to the end of the list.



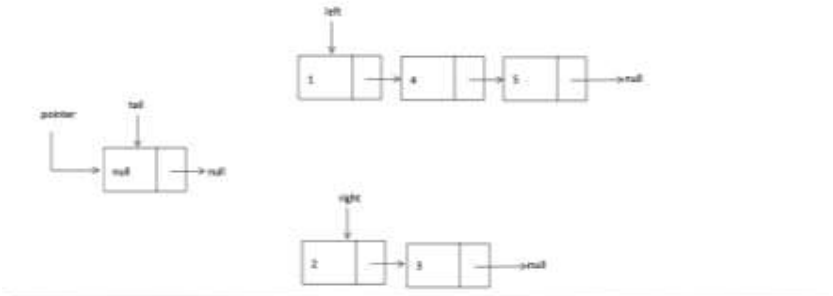

```
cityNode middle = getMiddle(start);
        cityNode nextOfMiddle = middle.next;
        middle.next = null;
```

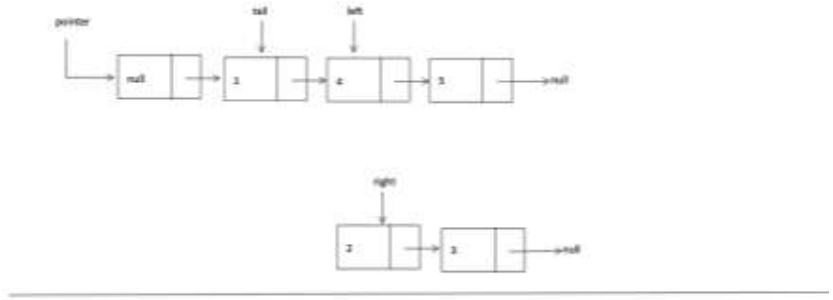
Two pointers, ‘left’ and ‘right’ are then initialized to recursively call on the ‘mergesort’ method. Left calls merge sort from the start node, while right calls on merge sort from ‘nextOfMiddle’.

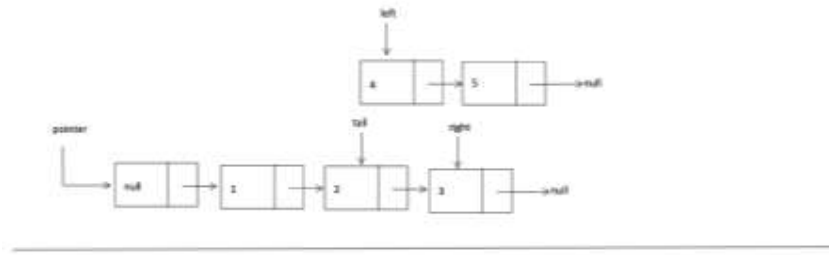
```
cityNode left = mergeSort(start); // Sort the left half
        cityNode right = mergeSort(nextOfMiddle);
```

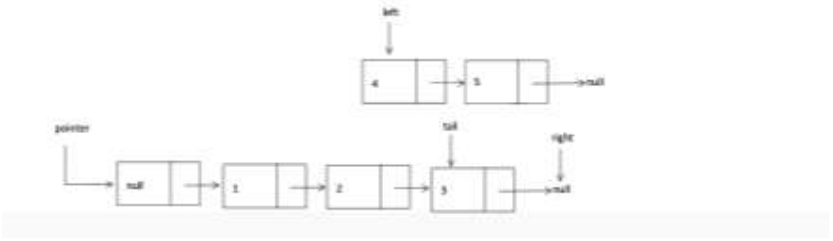
A third pointer, ‘sortedList’, then calls on the merge method, passing in the left and right nodes.
Within the merge method, two pointers, ‘pointer’ and ‘tail’, are created, with ‘tail’ initially pointing to ‘pointer’.

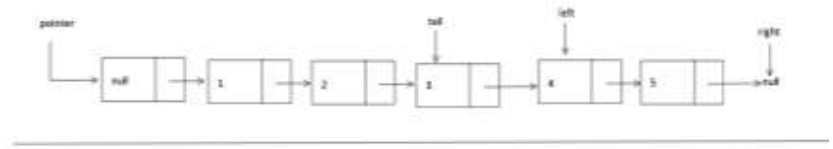
The method enters a while-loop, which runs as long as both the left and right nodes are not null. Within this loop, the method compares the current left and right node’s latitude, to determine which node to append to the ‘tail’ of the sorted list. If the left node’s latitude is smaller than the right node’s latitude, then the next of tail will be the left node, and left will point to left’s next. If the right node’s latitude is smaller than the left node’s latitude, then the next of tail will be the right node, and right will point to right’s next. Once appended, ‘tail’ is updated to point to the newly added node.
After the while-loop has terminated, the method checks which condition was met; if left is null, then the current right node will be appended to the tail, and if right is null, the current left node will be appended to the tail.











while (left != null && right != null) {

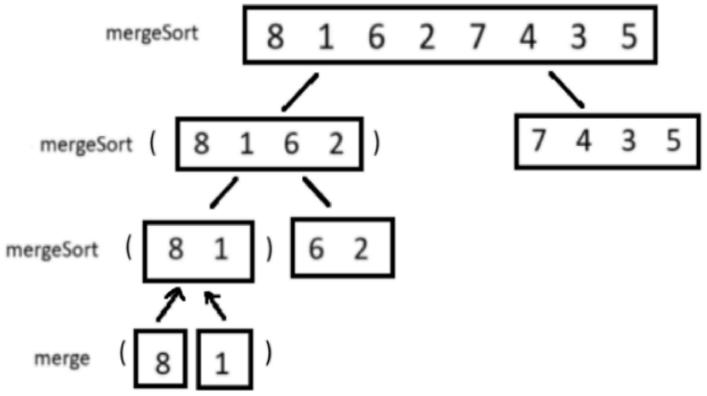
```
  if (left.city.lat() < right.city.lat()) {
      tail.next = left;
      left = left.next;
  } else {
      tail.next = right;
      right = right.next;
  }
  tail = tail.next;
}
if (left == null) {
        tail.next = right;
} else {
        tail.next = left;
}
return pointer.next;
```

Finally, the method returns the next node after pointer, representing the head node of the merged list.

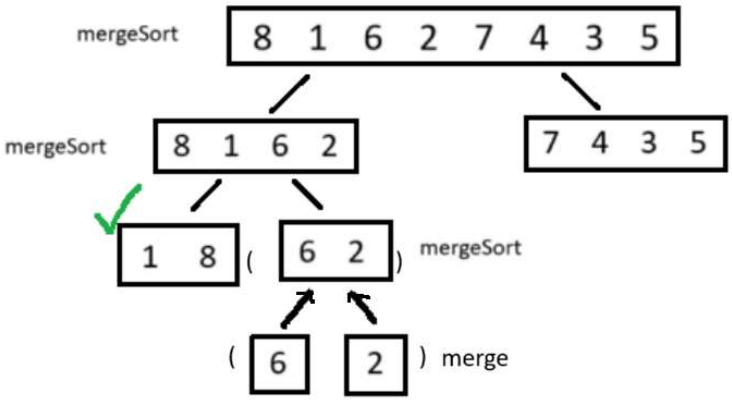
This node is then returned in the merge sort method.

Merge sort implemented with arrays:

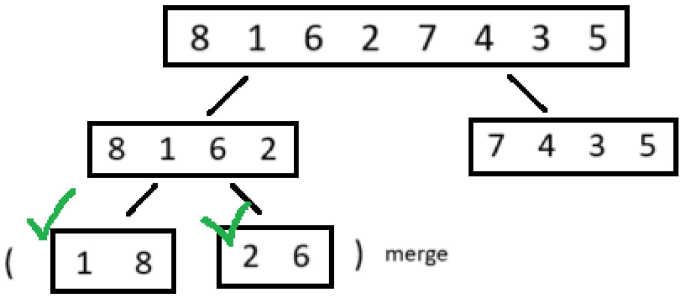
1. Divide the main array into two sub-arrays (left and right respectively). Continue doing so until the left sub-array reaches 1 element only. After that, start merging and sorting the left side.



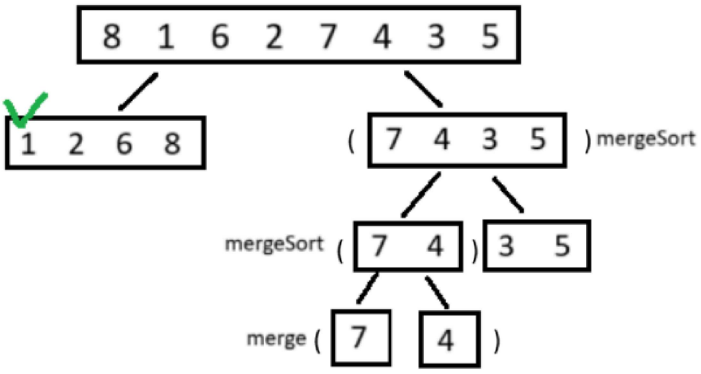
2. Do the same with the right side of the left subarray.



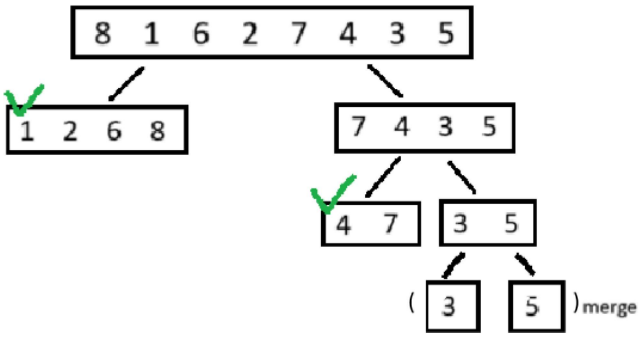
3. Merge and sort these two sub-arrays together.



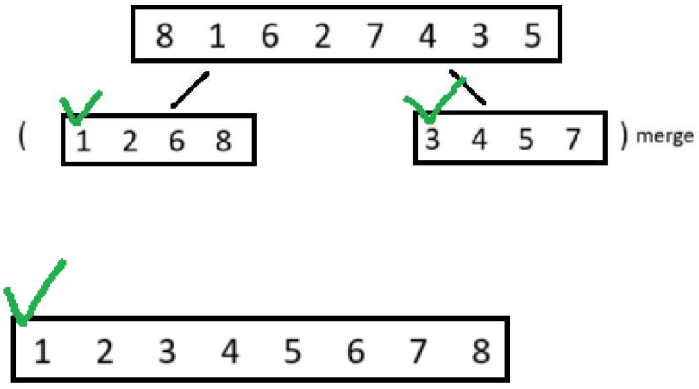
4. When the left subarray is done, start dividing the right subarray. Start again with the left side.



5. After that, do the same with the right side of the right subarray.



6. Finally, we sort and merge the last left and right subarrays back to the main array.



The merge sort method takes in the ArrayList of 'Cities' object as a parameter.
In this method, we determine the size of the array for organization purposes and calculate the midpoint to divide the array into two halves.
int size = cities.size();
int middle = size / 2;

We create a condition to stop dividing the array when it reaches one element.
Two subarrays, left and right, are created to hold the elements from the original array. We use a loop to distribute the elements into these subarrays.
We then recursively call the 'mergeSort' method with the 'left' subarray first, dividing the array until it reaches the base case (if (size <= 1) return). So we can to the subsequent ‘right’ subarray.

```
ArrayList<Cities> left = new ArrayList<>();
        ArrayList<Cities> right = new ArrayList<>();
        for (int i = 0; i < size; i++) {
```

```
        if (i < middle) {
            left.add(cities.get(i));
        } else {
            right.add(cities.get(i));
        }
    }
```

Then we call the merge method.
The merge method takes three parameters: left subarray, right array and the main 'cities' array.
Here we create another two integer variables to keep track of their traversals.

Three integer variables, ‘i’ for the main array, ‘l’ for the left subarray, and ‘r’ for the right subarray, are initialized to zero.
We iterate through both the left and right subarrays simultaneously until they are fully traversed.
During each iteration, we compare the latitudes of the current elements from the left and right subarrays. If the latitude of the element from the left side is smaller, it is placed in the merged array, and the corresponding indices are incremented.
Otherwise, the element from the right side is placed in the merged array.

```
int i = 0, l = 0, r = 0;
    while (l < leftSize && r < rightSize) {
        if (left.get(l).lat() < right.get(r).lat()) {
            cities.set(i, left.get(l));
            i++;
            l++;
        } else {
            cities.set(i, right.get(r));
            i++;
            r++;
        }
    }
```

After sorting the elements from both subarrays, if there are any remaining elements in either the left or right subarray, we iterate through them and add them to the merged array.

```
    while (l < leftSize) {
        cities.set(i, left.get(l));
        i++;
        l++;
    }
    while (r < rightSize) {
```

```
            cities.set(i, right.get(r));
            i++;
            r++;
        }
```

Once all elements are merged and sorted, the main cities array is updated accordingly.

b.

As mentioned in the first exercise, the merge sort consists of constantly dividing the list into halves until individual elements are reached, so that those elements can be merged back to a sorted list. This fundamental process does not change regardless of the order of the elements in the list.
For that reason, the number of merges required to merge sort a list remains the same no matter if the data is randomly sorted beforehand, because merge sort time complexity is always (n log(n)), where 'n' is the number of elements in the array.

In java, we calculated n (log n) with the implementation of the following code in ArrayList<Cities>:

```
double result = Math.log(cities.size()) / Math.log(2);
```

and LinkedList:

```
public double mergeCount(cityNode head) {
   cityNode temp = head;
   int r = 0;
   while (temp != null) {
      temp = temp.next;
      r++;
   }
   return Math.log(r) / Math.log(2);
}
```

c.

When implementing the merge sort algorithm for the latitude and longitude pairs, we used the haversine formula to calculate the distances. The Haversine distance formula is a reliable method used to calculate the shortest distance between two points on the surface of a sphere. When working with latitude and longitude of cities from the Earth, the Haversine formula gives a more accurate result in distance calculations. We opted to use Null Island as the reference point for the ordered list.

Null Island is situated where the prime meridian and equator intersect, meaning both the latitude and longitude is at zero degrees. The order of the sorted list is based on the distance from the coordinates at this point, which is calculated as the cities are stored in the linked list.

The following code snippet is the method used for calculating the distances.

```
double haversine (double val){
        return Math.pow(Math.sin(val / 2), 2);
}


double calculateDistance (double endLat, double endLng) {
   int earth_radius = 6371;
   double startLat = 0;
   double startLng = 0;
   double dLat = Math.toRadians((endLat - startLat));
   double dLong = Math.toRadians((endLng - startLng));
   startLat = Math.toRadians(0);
   endLat = Math.toRadians(endLat);
   double a = haversine(dLat) + Math.cos(startLat) * Math.cos(endLat) *
haversine(dLong);
   double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
   return earth_radius * c;
}
```

The main difference in the code is in the merge method, where the if-else statement within the while loop compares the left and right node's haversine distance instead of latitude.

```
 while (left != null && right != null) {
            if (left.city.haversineDistance() < right.city.haversineDistance()) {
                tail.next = left;
                left = left.next;
            } else {
                tail.next = right;
                right = right.next;
            }
```

```
    tail = tail.next;
  }
  tail.next = (left != null) ? left : right;
  return pointer.next;
```

Quick Sort

The quick sort algorithm, same as the merge sort, uses a divide-and-conquer technique. By using a chosen pivot point, the algorithm divides the array into two subarrays where the elements are organized based on their value. All the elements less than the pivot will be grouped into one subarray, and the elements equal to or greater than the pivot will go into another subarray. The pivot is then be placed at its correct index in between the two subarrays. This process recursively divides and sorts the array until each element is correctly placed at their right index. The outcome will be an ordered list in an ascending order.

In terms of space complexity, the algorithm typically requires O(n log(n)) space on average. The space is primarily utilized for the call stack during the recursive partitioning process. As for time complexity, quick sort has an average-case and best-case time of O(n log(n)). In a worst-case scenario, the time complexity increases to O(n^2), which in most cases happens when the array is fully or almost sorted. The average-case performance still makes the algorithm a preferred choice to efficiently sort large data sets.

a)

We chose to use the last element of the sequences as the pivot points in this algorithm.

QuickSort.*quickSort*(lat, 0, n - 1);

The quick sort method takes three parameters; the array to be sorted, the lowest possible index, being 0 when the entirety of the array is to be sorted, and the highest index, ‘n-1’ where ‘n’ is the size of the array. Initially, the method compares the low and the high index to make sure the given array has more than two elements. The method will continue to run as long as there are elements left to be sorted.

```
static int partition(ArrayList<City> arr, int low, int high) {
        double pivot = arr.get(high).lat();
        int i = (low - 1);
      for (int j = low; j <= high - 1; j++) {
        if (arr.get(j).lat() < pivot) {
                i++;
                swap(arr, i, j);
```

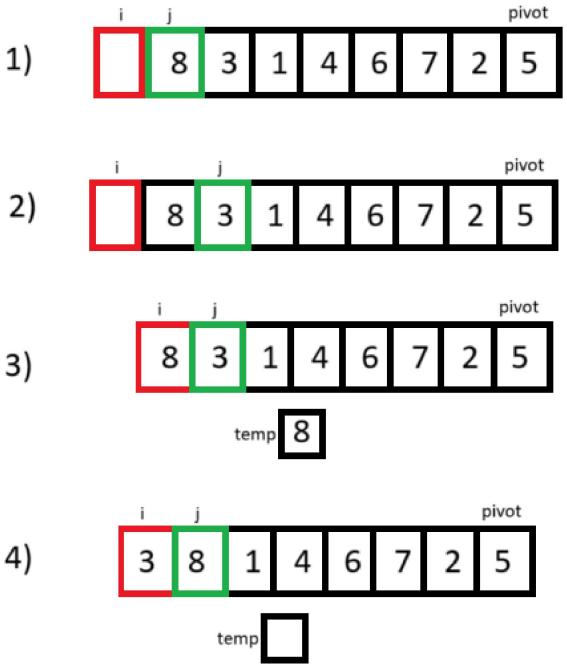
```
        }
         }
         swap(arr, i + 1, high);
         return (i + 1);
    }
```

The partitioning method takes the same parameters as the quick sort and contains two local variables. The pivot point, which in this case is the value of the highest index, and an integer ‘i’, used to sort the elements into their correct indexes. We will traverse through the array with a for loop with the integer ‘j’ to compare all the elements to the pivot. If the current element is smaller than the pivot, then ‘i’ will increment by one and the elements at index ‘i’ and index ‘j’ will swap places. The element will stay at the same index if it is greater than the pivot. This will go on until all the elements have been compared to the pivot, ending with all the smaller elements to the left of the array and all the greater elements to the right. The last swap consists of the element at index i and the pivot so that the pivot is at its right place between the lower and higher values. When the method is done it will return the index of the pivot.

```
static void swap(ArrayList<City> arr, int i, int j) {
        City temp = arr.get(i);
        arr.set(i, arr.get(j));
        arr.set(j, temp);
        }
```

Swapping of the elements during the partitioning process is done in a separate method. This method takes three parameters as well, the array and the indexes of the two elements which need to be swapped. A temporary variable is used to hold the values of one of elements we want to swap, in this case the value of the element at index ‘i’. With the value of the first element stored in the temporary variable, we are free to swap the value of the element at index ‘j’ to index ‘i’. The value of index ‘i’ is then assigned to the index ‘j’.

Here we have made a simple representation on how the algorithm stores and swaps the elements during the swap method of the quick sort:



From this point, ‘j’ will continue to increase to inspect the next element and compare it to the pivot point, with ‘i’ following if the value of the chosen element is less than the pivot. The algorithm will continue doing so until the array is fully traversed and the pivot is placed at its correct position, with the smaller elements to the left and the greater elements to the right.

```
static void quickSort(ArrayList<City> arr, int low, int high) {
        if (low < high) {
        int partition = partition(arr, low, high);
        quickSort(arr, low, partition - 1);
        quickSort(arr, partition + 1, high);
        }
    }
```

After the partitioning and swapping is complete, we will use the return value of the partition method to split the elements into two subarrays. The first subarray will consist of the smaller elements, with the biggest index given being partition - 1 and the second subarray containing the bigger elements starting from the index of partition + 1. The new partitions are then used to repeat the steps of partitioning and swapping, and the method will continue to recursively divide the array into two arrays and pass those in as arguments for the quick sort until base case is reached. In a quick sort algorithm, the base case is reached when the starting index is greater than or equal to the end index. This means that the array has less than two elements and no sorting is required.

b)

The number of comparisons required to sort a list using quick sort can vary depending on the arrangement of the elements within the array. In both the best and average cases, the time complexity remains (n(log n)). The best case occurs when the pivot consistently divides the array into precisely equal halves at each step, while the average case involves dividing the array into approximately balanced partitions.

However, if the array is randomly ordered in a manner where it is sorted or nearly sorted, then the pivot selection will consistently result in highly unbalanced partitions at each step, This imbalance leads to a greater number of comparisons, and consequently a worst-case time complexity of O(n^2).
In the context of our data set, 'worldcities', it aligns with the average case scenario. Making the number of comparisons (n (log n)), where n is the size of the array.
In summary, the number of comparisons can vary depending on how the array is ordered beforehand. For the 'worldcities' data set, the number of comparisons is calculated to be 913,159.

c)

As in the merge sort algorithm for the latitude and longitude pairs, we used the haversine formula to calculate the distance. This quick sort algorithm presented here follows approximately the same steps as the former quick sort algorithm explained, with the biggest difference lying in the partition method. When implementing haversine in the partitioning method of the sorting, the comparing value becomes the distance from the chosen pivot's coordinates to the reference point. The remaining elements are then compared against this calculated distance.

```
static int partition(ArrayList<City> arr, int low, int high) {
        double pivot = Haversine.calculateDistance(arr.get(high).lat(),
arr.get(high).lng());
        int i = (low - 1);

        for (int j = low; j <= high - 1; j++) {
        if (Haversine.calculateDistance(arr.get(j).lat(), arr.get(j).lng() < pivot) {
        i++;
        swap(arr, i, j);
        }
        }
        swap(arr, i + 1, high);
        return (i + 1);
 }
```

References:

https://www.geeksforgeeks.org/merge-sort/

https://www.geeksforgeeks.org/quick-sort/

https://www.programiz.com/dsa/merge-sort

https://www.javatpoint.com/merge-sort

https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-quick-sort/

Data Structures and Algorithms in Java, Fourth Edition.