

Course code: PG4200

Course name: Algoritmer og Datastrukturer

Submission deadline: 07.05.2024

Group number: 1

Candidates: 6, 49, 115, 129, 144.

Algorithms and Data Structures Exam: Report

Kristiania University College



Spring 2024

This assignment is written as part of the education at Kristiania University College.

The university is not responsible for the methods, results, conclusions or recommendations included in the assignment.

Table of contents

Introduction.....	2
Project and class structure.....	2
/app.....	3
/common.....	3
/problem1_MergeSort.....	4
/problem2_Quick sort.....	5
How to run and use the program.....	5
Through the Main Application.....	5
By running each task separately.....	6
Problem 1: Merge Sort.....	6
a. Implementation with only latitude.....	6
Main method.....	6
Dividing.....	7
Merging.....	8
b. Implementation for comparing number of merges.....	10
c. Implementation with both latitude and longitude.....	11
1. Euclidean distance by using euclidean formula.....	12
2. Geographical Points by using haversine formula.....	13
Problem 2: Quick Sort.....	16
a. Implementation with only latitude.....	16
Main method.....	16
Dividing.....	16
Partitioning and comparing.....	18
b. Implementation for comparing number of comparisons.....	20
c. Implementation with both latitude and longitude.....	23
1. Euclidean distance by using Euclidean formula.....	23
2. Geographical points by using Haversine formula.....	23
Sources.....	25

Introduction

In this section, you will find the project and class structure that explains what each class does and why we chose to set up the project in the way we did. You will also find a short user manual for using the program as intended.

Project and class structure

This is what the project looks like when opened in IntelliJ. Below the image, the classes and code will be explained in further detail.

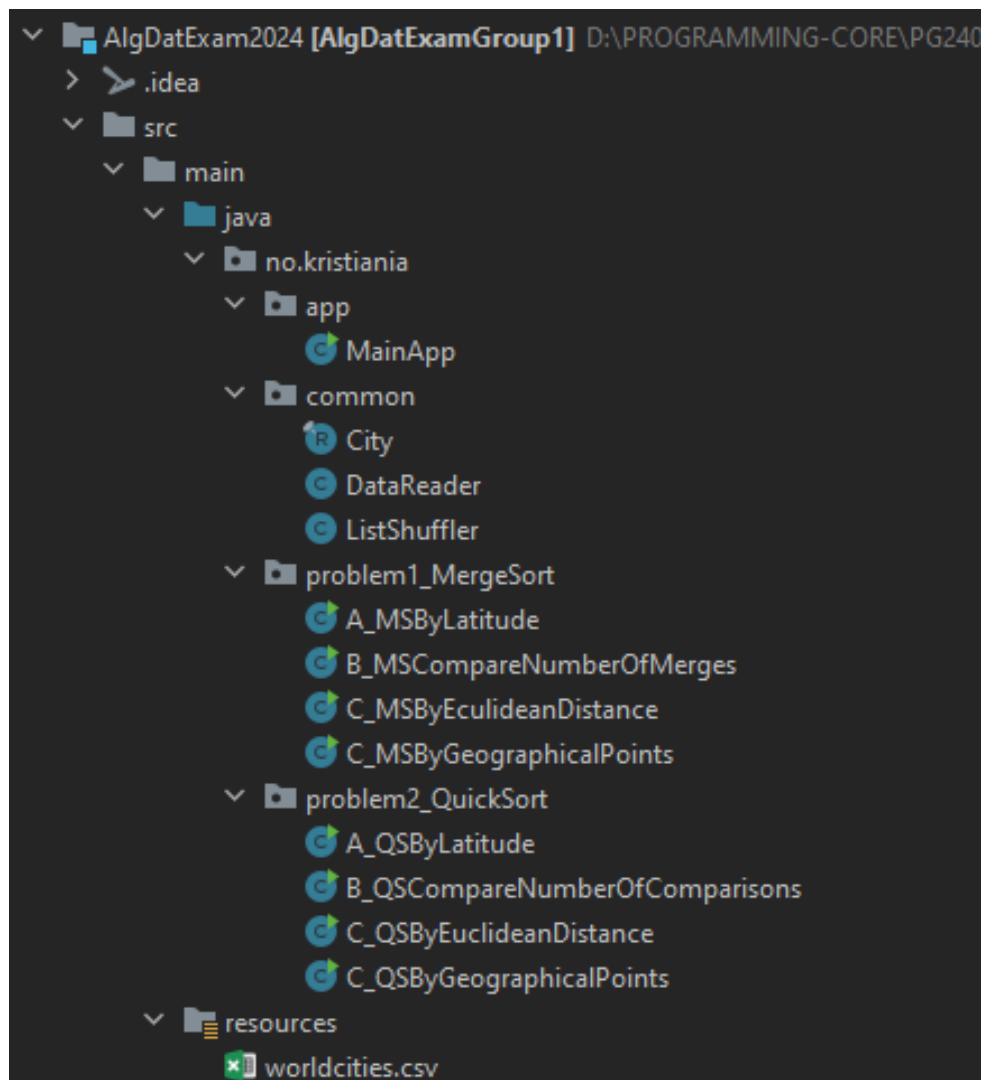


Figure 01: Project and class structure.

/app

MainApp

The “MainApp” class is the typical main class in Java for this project and, as the name suggests, is intended to run as the primary application for the entire program. This class contains a “Scanner” instance to read user input and a “while” loop that controls the program flow based on user choices. The program will continue to run until the “running” variable is set to “false”.

The program displays a menu with various options for executing solutions to the different subtasks under each problem in the exam assignment. Users are prompted to enter a number to select a task, and this input is read by “`scanner.nextInt()`”. A “switch” statement processes the user's choice and triggers the corresponding class's “main” method based on that selection. After completing each task, the program prompts the user to press the enter key to return to the main menu.

/common

DataReader

This class is responsible for reading and processing city data from a CSV file and converting it into a list of “City” objects. It initializes with a file path and uses a “BufferedReader” to read the file efficiently, skipping the header line. The main method, “readCitiesFromCSV”, reads each line of the CSV and splits it into components. It then parses the data into “City” objects, extracting each city's name, latitude, and longitude, and converts these to appropriate data types. The method includes error handling for parsing issues and insufficient data in rows. It returns a list of “City” objects, ensuring that each entry has valid latitude and longitude values.

City

The “City” class, structured as a Java record, specifies the essential attributes (“name”, “latitude”, and “longitude”) that a City object holds. This structure is directly utilized by the “DataReader” class to parse and convert data from a CSV file into “City” objects efficiently. The record's immutable nature ensures that once a “City” object is created, its state cannot be altered, promoting safe use across the application.

ListShuffler

The “ListShuffler” class provides a functionality to randomly shuffle arrays, specifically designed to handle lists of geographical coordinates as in the dataset we are working with. Using the Fisher-Yates shuffle algorithm (GeeksforGeeks, 2022), it ensures an efficient and fair randomization of the elements in the array. The class's static method “shuffleLists” uses the Java “Random” class to determine random swap positions for elements within the array, effectively shuffling the list of latitudes.

/problem1_MergeSort

This folder contains the solutions to all the subtasks for Problem 1. Merge Sort. There are two solutions for task C. Initially, we started by solving the task using the Euclidean distance calculated with the Euclidean formula, before realizing that using geographical points calculated with the Haversine formula provided more appropriate values to work with.

In the code for the assignments, we have focused on readability over object-oriented and modular code. This is because it should be clear to the evaluator what we have planned and how we have proceeded with each task.

/problem2_Quick sort

This folder contains the solutions to all the subtasks for Problem 2. Quick Sort. Similar to `"/problem1_MergeSort"`, it includes two solutions for task C, for the same reasons as described earlier, with the implementations being focused on readability here as well.

How to run and use the program

There are two ways to run the solutions. You can either choose to run them through the Main Application (`"MainApp"`), or run each task separately. Below are instructions for both.

Through the Main Application

1. Open the project folder in IntelliJ so that it appears as shown in the "Figure 01: Project and class structure."
2. Open the file `"MainApp"`-class in the `"app"` folder, and run it.
3. You will now see a welcome message and a numbered list with menu options in the terminal. At the bottom, the program prompts you to choose which task you want to run. Enter the corresponding number to the task you wish to execute, and press enter.
4. The program will now have printed the result of the task you chose to run. Once you've finished reading the content, press the enter key again to return to the main menu.
5. If you want to run another task, return to step 3 and repeat the process. If you wish to terminate the program, choose the option `"9. Terminate the program"`, press enter, and the program will exit.

By running each task separately

1. Open the project folder in IntelliJ so that it appears as shown in the "Figure 01: Project and class structure."
2. Navigate to either `"/problem1_MergeSort"` or `"/problem2_QuickSort"`. Here, you will find classes with names representing the task they solve.
3. Choose a task and run it.

Problem 1: Merge Sort

a. Implementation with only latitude

Main method

From our "main" method, the latitudes from the CSV file are extracted into an array to perform a Merge Sort algorithm.

```
public static void main(String[] args) {  
    String csvFilePath = "src/main/resources/worldcities.csv";  
  
    DataReader csvReader = new DataReader();  
    List<City> cities = csvReader.readCitiesFromCSV(csvFilePath);  
  
    double[] latitudes = new double[cities.size()];  
    for (int i = 0; i < cities.size(); i++) {  
        latitudes[i] = cities.get(i).latitude();  
    }  
  
    sort(latitudes);  
  
    System.out.println("\nSorted latitudes:");  
    for (double latitude : latitudes) {  
        System.out.printf("%.4f\n", latitude);  
    }  
}
```

Figure 02: main method in "A_MSByLatitude".

A Merge Sort is a sorting algorithm that uses a divide-and-conquer approach to sort an array: divide, conquer and merge.

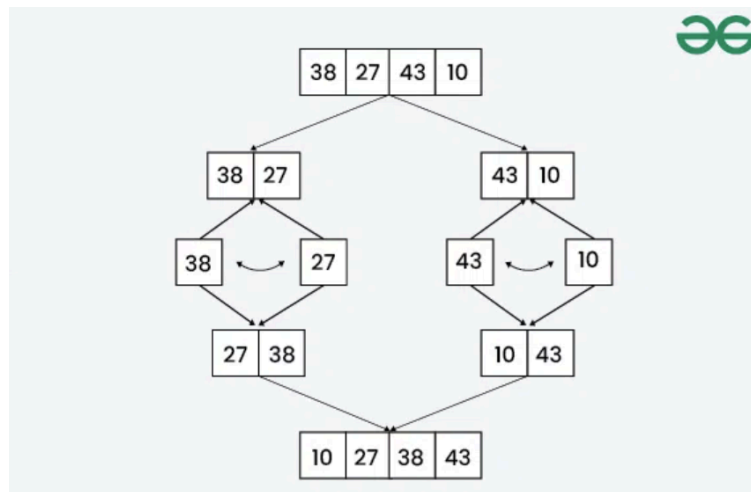


Figure 03: Divide and Conquer Visualisation (GeeksforGeeks, 2024, 30.04).

The “sort” method below first checks to see if the array is “null” or if its length is less than or equal to one. If the array is equal to “null” there is nothing to sort. However, if the array contains one or no elements there will be no need to perform any sorting operations, as a single element or an empty segment is already inherently sorted. If neither of these conditions are true, the method “mergeSort” is called.

```
public static void sort(double[] arr) { 1 usage
    if (arr == null || arr.length <= 1) {
        return; // Array is already sorted or empty
    }
    mergeSort(arr, left: 0, right: arr.length - 1); // Start merge sort
}
```

Figure 04: sort method in “A_MSByLatitude”.

Dividing

Within the “mergeSort” method, we start with a condition to check if left is less than right. If this condition returns “false”, it implies that the array contains one or zero elements, and therefore is considered already sorted. This ensures that the method only processes segments of the array that are larger than one element. If “true”, the method calculates the midpoint which is used to divide the array into two

halves. Then, recursive calls are made to sort the left and right halves of the array. The process of dividing the array continues until the base case of the recursive algorithm is reached. This is when our original array is divided into single-element arrays, and “left < right” returns False.

```
private static void mergeSort(double[] arr, int left, int right) { 3 usages
    if (left < right) {
        int mid = left + (right - left) / 2; // Calculate the middle index
        mergeSort(arr, left, mid); // Sort the left half
        mergeSort(arr, left: mid + 1, right); // Sort the right half
        merge(arr, left, mid, right); // Merge the sorted halves
    }
}
```

Figure 05: mergeSort.

Recursion is used to reduce the complexity of the sorting. This is achieved by continually breaking down the problem, creating an ordered list containing all city latitudes, into smaller problems by sorting different parts of the array. With each division, the depth of the recursion increases.

Merging

The “merge” method below, in “Figure 06: merge”, is crucial for the merging process within our Merge Sort algorithm. It combines two sorted subarrays into a single sorted subarray of our original array.

First, the method calculates the size of the two subarrays, “left” and “right”. It then creates temporary arrays to hold the elements of these two subarrays. The subsequent two for-loops copy the elements from the original array into these temporary subarrays. The merging of the left and right subarrays follows, where the method compares elements from each subarray and places the smaller element into the original array. This process continues until all elements from one subarray have been placed.

After this merging step, it's possible that elements remain in either the left or the right subarray because they were already in the correct position. The method then copies these remaining elements directly back into the original array, ensuring that no elements are omitted. While the “merge” method efficiently merges the two subarrays during each call, the entire array only achieves full sorting after all recursive divisions and subsequent merging steps have been completed throughout the “mergeSort” process as shown in “Figure 05: mergeSort”.

```
private static void merge(double[] arr, int left, int mid, int right) { 1 usage
    int n1 = mid - left + 1; // Size of the left subarray
    int n2 = right - mid; // Size of the right subarray

    double[] L = new double[n1];
    double[] R = new double[n2];

    for (int i = 0; i < n1; i++) {
        L[i] = arr[left + i];
    }
    for (int j = 0; j < n2; j++) {
        R[j] = arr[mid + 1 + j];
    }

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

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

Figure 06: merge.

b. Implementation for comparing number of merges

In the “B_MSCompareNumberOfMerges” class, we have implemented functionality to track the number of merge operations performed during the execution of the Merge Sort algorithm. A static variable, “mergeCount”, is used for this purpose, and it is incremented each time the “merge” method is called, which merges two sorted subarrays into a single sorted subarray.

During a Merge Sort, we continue to split the array until each subarray contains only one single element. The process of division does not depend on the order of the elements, but on the total number of elements within the array we are trying to sort. The number of recursive levels is determined by the number of divisions needed until we are left with only single element arrays.

After the division process, every subarray is merged. During this process the elements are first compared to each other before merging. This part of the sorting algorithm also remains consistent, and is only affected by the size of the array. Each recursive call is determined by the number of subarrays needed to merge in order to create the new sorted array. The number of subarrays is determined by the number of elements to be sorted, not by the original order.

To effectively demonstrate the impact of shuffling on the Merge Sort process, the task is performed in a manner that allows the user to easily compare the number of merges conducted before and after shuffling the array elements. This is achieved by sorting the array, recording the number of merges, shuffling the array, and then sorting it again. The numbers of merges from both sortings are then displayed, followed by a message indicating whether the counts were equal or differed, thus providing direct feedback on how the shuffling influenced the sort efficiency.

```
Number of merges before the shuffle: 47867
Shuffling the list...
Number of merges after the shuffle: 47867

Number of merges was equal in both cases.

Process finished with exit code 0
```

Figure 07: Terminal output for Problem 1, task b.

We can conclude that the number of merges needed to sort an array is not determined by the initial order of the elements. Whether or not we shuffle the elements around, the number of merges performed still remains constant. For an array with N elements, we observe that the total number of merges completed was always $N - 1$. This is one of the reasons why Merge Sort is considered to deliver a predictable time complexity, which is always $O(n \log n)$ (GeeksforGeeks, 2024).

c. Implementation with both latitude and longitude

Each city in the dataset holds a latitude and a longitude value that will be used to solve this task. Sorting a pair of values, such as 60.5816 and 169.05 is difficult without either assigning them a separate key or taking a shortcut and simply add them together ($60.5816 + 169.05$) and then sort them by value. However, we have decided that this approach is not suitable to solve this task.

Before proceeding, it's important to ensure the 47,868 pairs of numbers in the dataset has any relevance to a real world application. To verify that these values represent points on a flat surface, we assume they are Cartesian coordinates. We confirm this assumption by plotting the coordinates on an X/Y-graph as a scatter chart.

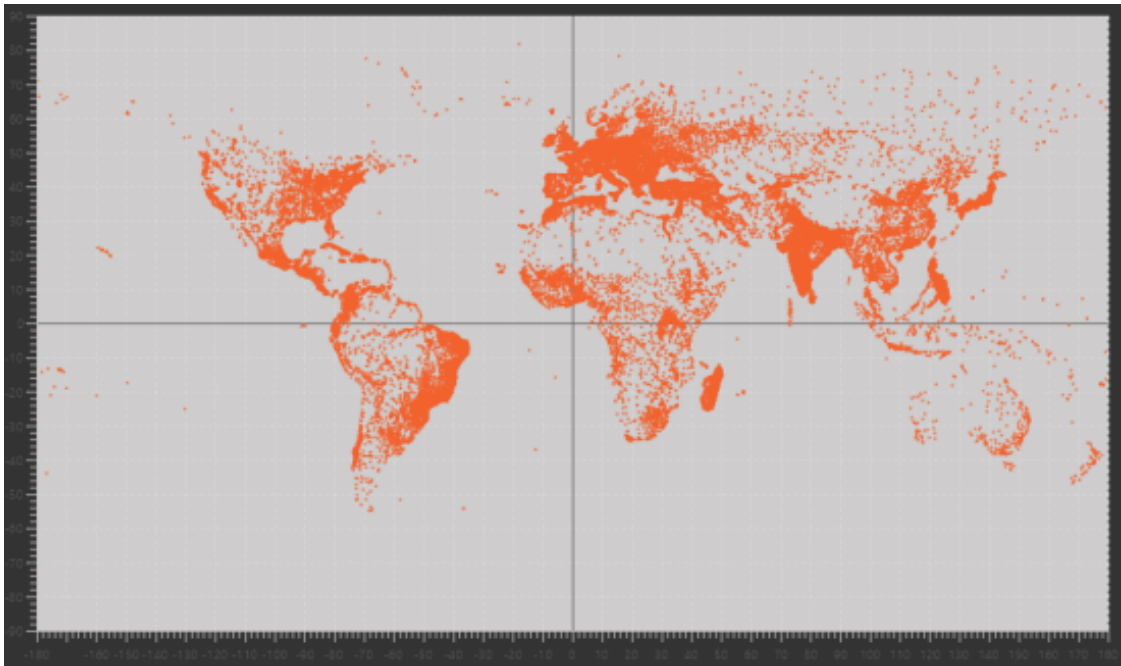


Figure 08: Visualizing all the coordinates on an X/Y-graph as a scatter chart.

1. Euclidean distance by using euclidean formula

We originally considered using the Euclidean distance formula to calculate the distance from Origo using longitude and latitude as coordinates (Wikipedia, 2024). This would calculate a number suitable for sorting algorithms, but it would only be relevant in a geometric context and would not account for the curvature of the Earth. We choose to use the Haversine formula instead, because it determines the great circle distance between two points on a sphere based on longitude and latitude. This method considers the curvature of the Earth and produces sortable numbers (Wikipedia, 2024).

As one can tell from the output, the result is a sorted list by a geometrical number.

```
Euclidean distance: 188,9513
Euclidean distance: 189,1235
Euclidean distance: 190,0741
Euclidean distance: 191,0639
Euclidean distance: 193,1002

Sorting completed based on Euclidean geometrical distance

Total number of merges: 47867
```

Figure 09: Sorted list by a geometrical number.

2. Geographical Points by using haversine formula

```
Charlotte Amalie EUQ :: 67.473968607234 HAV :: 7370.807110795073 VIN :: 7376.222301567278
```

Figure 10: Output of three different mathematical formulas.

Figure 10 contains a comparative output of three different mathematical formulas that could be used to solve the assignment. The Euclidean formula produces a value that is useful for sorting, but lacks practical value. However, both the Haversine and Vincenty formula can return the distance from point Origo in kilometers, providing a more functional value. We choose the Haversine formula instead of Vincenty because it is accurate enough for the purpose of this task, as it produces a relevant number. We can quickly confirm this with a comparison with Google Maps. While one could argue for the greater accuracy of the Vincenty distance formula, it is also significantly more complex and out of scope for the current tasks, seeing as the marginal difference is about 6 kilometers.

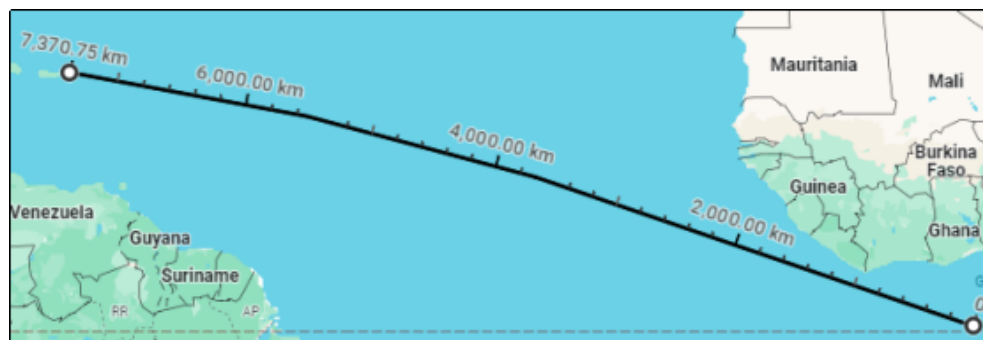


Figure 11: Screenshot from Google Maps.

```
haversineFormula[i] = haversineDistance( latOne: 0.0, lngOne: 0.0, cityLat[i], cityLng[i]);
```

Figure 12: haversineFormula.

Now that we have established the distance measure, style and cause, it is possible to sort the cities by real distance from point Origo.

Depending on preference, this can be performed in an ordered list either ascending or descending. By solving the problem in this fashion we also avoid added complexity down the line, after the cities have been sorted and indexed. It will directly enable the implementation of basic binary search functionality without any further conversions.

The main difference between assignments “a”, “b”, compared to “c” is how the data is handled prior to sorting. In “a” and “b”, the algorithms directly use the data sourced from the document. In contrast, assignment “c” processes the longitude and latitude values through a mathematical formula, specifically the Haversine formula, to compute distances before sorting.

```

private static double calculateHaversineDistance(double lat1, double lon1) { 2 usages
    double lat1Rad = Math.toRadians(lat1);
    double lon1Rad = Math.toRadians(lon1);

    double lat0Rad = Math.toRadians(0.0); // Assume reference point is the equator and prime meridian (0, 0)
    double lon0Rad = Math.toRadians(0.0);

    double deltaLat = lat1Rad - lat0Rad;
    double deltaLon = lon1Rad - lon0Rad;

    double a = Math.sin(deltaLat / 2) * Math.sin(deltaLat / 2) +
        Math.cos(lat0Rad) * Math.cos(lat1Rad) *
        Math.sin(deltaLon / 2) * Math.sin(deltaLon / 2);
    double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));

    return EARTH_RADIUS_KM * c; // Return the distance in kilometers
}

```

Figure 13: calculateHaversineDistance method in “C_MSByGeographicalPoints”.

The output of the Haversine formula calculates distances, which are then organized using the “mergeSort” algorithm into an ascending list. This sorts each city based on its kilometer distance from point Origo. The process also tracks the number of merge operations for insight into the sorting efficiency.

```

Geographical Point: 18776,6656
Geographical Point: 19063,9349
Geographical Point: 19215,1670
Geographical Point: 19224,6182

Sorting completed based on geographical kilometers from point Origo

Total number of merges: 47867

```

Figure 14: Output for C_MSByGeographicalPoints.

Problem 2: Quick Sort

a. Implementation with only latitude

Main method

From our “main” method, in “A_QSByLatitude”, the latitudes from the CSV file are extracted into an array to perform a Quick Sort algorithm. The “main” method prints out the sorted array to the console.

```
public static void main(String[] args) {
    String csvFilePath = "src/main/resources/worldcities.csv";

    // Create an instance of the AlgDat.common.CSVReader class
    DataReader csvReader = new DataReader();

    // Use the AlgDat.common.CSVReader instance to read cities data from CSV file
    List<City> cities = csvReader.readCitiesFromCSV(csvFilePath);

    // Extract latitudes into an array
    double[] latitudes = new double[cities.size()];
    for (int i = 0; i < cities.size(); i++) {
        latitudes[i] = cities.get(i).latitude();
    }

    // Create an instance of A2QuickSort
    A_QSByLatitude quickSort = new A_QSByLatitude();

    // Sort latitudes using quick sort
    quickSort.sort(latitudes);

    // Output the sorted latitudes
    System.out.println("\nSorted latitudes:");
    for (double latitude : latitudes) {
        System.out.printf("%.4f\n", latitude);
    }
}
```

Figure 15: main method in A_QSByLatitude.

Dividing

In Quick Sort, a pivot is used to split the array into two segments. Each partitioning step involves comparing every element with the pivot. The recursive nature of the sorting introduces a logarithmic factor, resulting

in an average time complexity of $O(n \log n)$. The worst-case scenario, with a complexity of $O(n^2)$, occurs when the largest array element is selected as the pivot. In such cases, the array fails to split evenly, and the pivot ends up at the end, leaving the bulk of the array unsorted and causing numerous recursive calls.

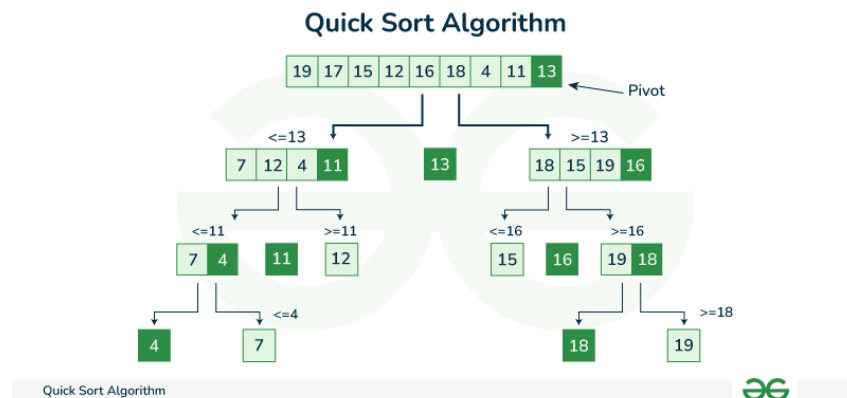


Figure 16: Quick Sort Algorithm (GeeksforGeeks, 2024 02.01).

Below, the Quick Sort is implemented with the median-of-three as an optimization technique for selecting the pivot. This is done to prevent the worst-case performance, typically encountered with sorted or nearly sorted arrays. As previously discussed, a poor pivot choice can result in worse performance, and potentially worst-case complexity. The performance of a Quick Sort is directly affected by the pivot, where the pivot potentially can make the sorting less efficient.

The method below finds the median of the three elements “a”, “b” and “c”. It sorts the three elements, and selects the middle as the pivot.

```
// Function to find the median of three numbers
double medianOfThree(double a, double b, double c){ 1 usage
    if ((a > b) && (a < c)){
        return a;
    } else if (b > c && (b > a)){
        return b;
    } else {
        return c;
    }
}
```

Figure 17: medianOfThree method in "A_QSByLatitude".

Partitioning and comparing

The "partition" method is implemented to rearrange the elements of the subarray. Pivot is chosen using the "medianOfThree" method shown above, and the pivot is placed at the end of the subarray, and therefore swapped with the last element.

The elements that are smaller than the pivot are placed to the left, and the greater elements to the right. The pivot is then placed at its correct position in the array. Since all the elements within the array are placed in relation to the pivot, the pivot will already be in its sorted and final position.

In the most efficient case of a Quick Sort, the recursion depth will grow logarithmically with the size of the array. This will result in a time complexity of $O(\log n)$. The $\log n$ value counts the number of times the array size n is divided by 2, until reaching 1. When reaching a subarray of 1 element, this will be the base case that stops further division (GeeksforGeeks, 2024) .

```
// Function to partition the array on the basis of pivot element
private int partition(double[] arr, int low, int high) { 1 usage
    double pivot = medianOfThree(arr[low], arr[(low + high) / 2], arr[high]);

    // Find the pivot index
    int pivotIndex = low;
    if (pivot == arr[(low + high) / 2]) {
        pivotIndex = (low + high) / 2;
    } else if (pivot == arr[high]) {
        pivotIndex = high;
    }

    // Swap pivot element with the last element
    double temp = arr[pivotIndex];
    arr[pivotIndex] = arr[high];
    arr[high] = temp;

    // Partition the array
    int i = low;
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            // Swap elements
            temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
            i++;
        }
    }

    // Move pivot element to the correct position
    temp = arr[i];
    arr[i] = arr[high];
    arr[high] = temp;

    return i;
}
```

Figure 18: partition method in “A_QSByLatitude”.

The sort method below is a recursive method that repeatedly partitions the array. After, it recursively sorts the elements of the subarrays before and after the partitioning index.

```
// Function to perform quicksort
private void sort(double[] arr, int low, int high) { 3 usages
    if (low < high) {
        // Partition the array and get the partitioning index
        int pi = partition(arr, low, high);

        // Recursively sort elements before and after the partition
        sort(arr, low, high: pi - 1);
        sort(arr, low: pi + 1, high);
    }
}
```

Figure 19: Quick Sort from “A_QSByLatitude”.

The “sort” method calls the array to be sorted, from the first element (index 0) until the last element of the array.

```
// Function to sort the array
public void sort(double[] arr) { 1 usage
    sort(arr, low: 0, high: arr.length - 1);
}
```

Figure 20: sort method from “A_QSByLatitude”.

b. Implementation for comparing number of comparisons

The time complexity and overall efficiency are directly impacted by the number of comparisons needed to sort the dataset. In the Quick Sort algorithm, this is determined both by the original order of elements, and the choice of pivot. To enhance the efficiency we therefore implement the median-of-three method. This method involves selecting three random elements from the array, and using the median of these three values as the pivot. The main goal of implementing this optimization technique is to reduce the probability of the pivot being the smallest or the greatest element of the array. With real world data, there is a possibility of the elements being partially or fully sorted. In these cases, if the pivot is set to always be the last element of the array, there might be a situation where the pivot is consistently poor throughout the sorting. If there are no elements greater than the pivot, the

pivot will not effectively divide the array. Instead it will leave the rest of the array to be sorted, and by each recursive call the array size is only decreased by one element. The large number of recursive calls can lead to a significant degrade in the algorithms performance, resulting in a time complexity of $O(n^2)$ (w3schools).

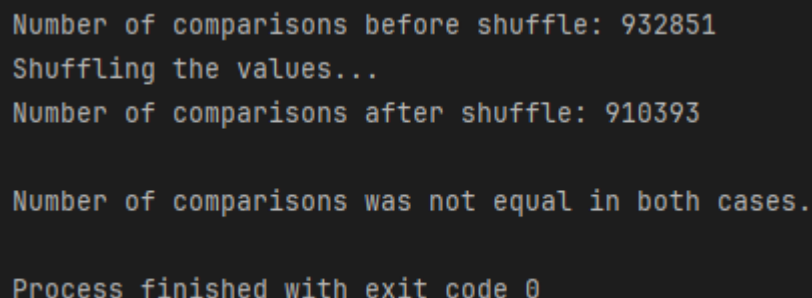
- **Best case** scenario in a Quick Sort is reached when the pivot consistently divides the array into two equal halves throughout the sorting process. This gives us a time complexity of $O(n \log n)$.
- **Average** time complexity of $O(n \log n)$, which occurs as long as the pivot divides the array into reasonably proportionate halves.
- **Worst case** scenario will result in a $O(n^2)$ complexity as a result of a poor pivot choice. This situation occurs when the pivot is always the smallest or the greatest element of the array, creating very unbalanced partitions.

The number of comparisons needed to sort the dataset changes when the order of the elements are randomly sorted. The reason for this is that when the pivot determines the number of comparisons and the order of the elements changes, the pivot element also changes. If the pivot is always chosen as the first or the last element of the array, the number of comparisons needed in a Quick Sort can dramatically change after the elements are randomly sorted. This is due to the many variations of possible sequences. When an optimization technique is implemented, like the median-of-three method, this will result in a more stable outcome. By using three samples from the array, the method helps choose a pivot that is more centered. The likelihood of experiencing the worst case scenario of $O(n^2)$ complexity is therefore reduced. However, even though the optimization is implemented and can help reduce the number of comparisons in average scenarios, the theoretical minimum number of comparisons needed is not reduced. The technique can not improve the asymptotic runtime beyond $\Omega(n \log n)$. This

shows that the performance can be improved, but the algorithms fundamental computational complexity limit can not be surpassed (Stanford University, 2018).

If we had not implemented the median-of-three optimization, and instead opted for a fixed pivot position, the outcome would be different. If the array is partially or already fully sorted, shuffling the elements around might improve the performance of the Quick Sort algorithm. Especially when handling large data sets there might be a higher possibility of the array already being sorted. If the pivot is to be at a fixed position, let's say always the last element, there is a higher risk of encountering the worst case scenario of $O(n^2)$. Therefore by randomly sorting the elements, there is a higher chance of the pivot dividing the array into reasonably proportionate halves.

Similarly to Problem 1 task b, we implement the solution in a separate class using a "try-catch" statement. Here, cities are read, sorted using Quick Sort, and the number of comparisons made before shuffling is printed out. This solution allows the user to quickly and concisely see the number of comparisons before shuffling is printed. Then, the list of cities is shuffled before being sorted again with the Quick Sort algorithm, and the number of comparisons after shuffling is printed out in the same manner as before. Finally, a message is displayed in the terminal informing the user whether the number is the same or not.



```
Number of comparisons before shuffle: 932851
Shuffling the values...
Number of comparisons after shuffle: 910393

Number of comparisons was not equal in both cases.

Process finished with exit code 0
```

Figure 21: Output from *B_QSCompareNumberOfComparisons*.

In conclusion; randomly sorting the array always changes the number of

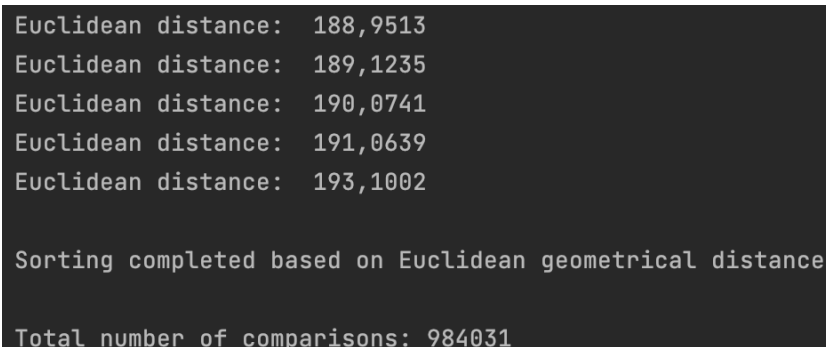
comparisons in a Quick Sort algorithm. This remains the same if opting for a median-of-three method, or setting the pivot at a fixed position. The pivot choice can drastically change the number of comparisons needed to perform a Quick Sort algorithm, and result in different time complexities - but the number of comparisons will still always differ.

c. Implementation with both latitude and longitude

See Problem 1 task c, for introduction to the paired coordinate sorting.

1. Euclidean distance by using Euclidean formula

The Euclidean formula only measures the distance between two points on an X/Y-graph (Wikipedia 2024, 17.03). The following image is the output from “C_QSByEuclideanDistance” using the Euclidean formula.

A screenshot of a terminal window with a dark background and light-colored text. The output shows five lines of Euclidean distance calculations, followed by a completion message and a total comparison count.

```
Euclidean distance: 188,9513
Euclidean distance: 189,1235
Euclidean distance: 190,0741
Euclidean distance: 191,0639
Euclidean distance: 193,1002

Sorting completed based on Euclidean geometrical distance

Total number of comparisons: 984031
```

Figure 22: Output from C_QSByEuclideanDistance.

2. Geographical points by using Haversine formula

The algorithm employed for implementing geographical points is the same that is used in “A_QSByLatitude”, which also includes an explanation of Quick Sort. Instead of using the latitude as the value for sorting, the output of the “calculateHaversineDistance” method is utilized.

The output of the Quick Sort algorithm using Haversine distance calculation produces an ascending list where each city is sorted based on its total kilometer distance from the reference point Origo. The total number of comparisons is printed as well.

```
Geographical Point: 18559,5048
Geographical Point: 18776,6656
Geographical Point: 19063,9349
Geographical Point: 19215,1670
Geographical Point: 19224,6182

Sorting completed based on geographical kilometers from point Origo

Total number of comparisons: 956048
```

Figure 23: Output from C_QSByGeographicalPoints.

Sources

GeeksforGeeks. (2022, December 19). Shuffle a given array using Fisher–Yates shuffle algorithm. Retrieved May 4, 2024, from:

<https://www.geeksforgeeks.org/shuffle-a-given-array-using-fisher-yates-shuffle-algorithm/>

GeeksforGeeks. (2024, January 2). Quick Sort in C. Retrieved May 3, 2024, from:

<https://www.geeksforgeeks.org/quick-sort-in-c/>

GeeksforGeeks. (2024, April 8). Merge Sort – Data Structure and Algorithms Tutorials.

Retrieved April 9, 2024, from:

<https://www.geeksforgeeks.org/merge-sort/?ref=lbp>

GeeksforGeeks. (2024, April 9). QuickSort – Data Structure and Algorithm Tutorials.

Retrieved April 10, 2024, from:

<https://www.geeksforgeeks.org/quick-sort/?ref=lbp>

GeeksforGeeks. (2024, April 30). Divide and Conquer Algorithm. Retrieved May 3, 2024, from:

<https://www.geeksforgeeks.org/divide-and-conquer/>

Stanford University. (2018, January 29). Algorithm Design and Analysis. Retrieved April 18, 2024, from:

https://graphics.stanford.edu/courses/cs161-18-winter/Recitations/section3win2018_withsols.pdf

W3Schools. (n.d.). DSA Quicksort. Retrieved April 17, 2024, from:

https://www.w3schools.com/dsa/dsa_algo_quicksort.php

Wikipedia. (2024, April 9). Haversine formula. Retrieved April 23, 2024, from:

https://en.wikipedia.org/wiki/Haversine_formula

Wikipedia. (2024, March 17). Euclidean distance. Retrieved April 22, 2024, from:

https://en.wikipedia.org/wiki/Euclidean_distance