Analysis of Merge sort and Quicksort Algorithms in Geographical Data

I. Introduction

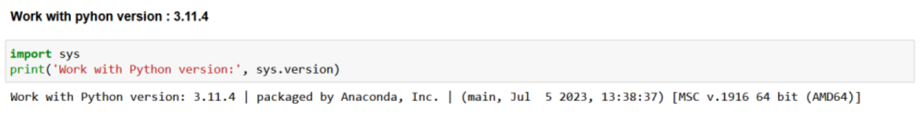
Sorting algorithms are valuable tools for structuring and organizing datasets, making the analysis and retrieval process quicker and efficient, especially for a large dataset, such as the one used in this report, making the process of finding valuable insights easier. There are 43 different sorting algorithms, however this report will focus on the two most popular and widely used, called Merge Sort and Quicksort. These are employed for the purpose of analyzing and understanding their algorithmic behavior in a particular geographical area. Additionally, we'll explore the implementation of changes when randomly arranging the data, to understand the complexity and the adaptability of the algorithms.

II. Data Source

The Basic World Cities Dataset, used in this report, is a compilation of the world's most populous places determined by the government agencies of the US, ensuring a foundation of accuracy and reliability. It includes all the populated areas worldwide, excluding all the neighborhoods located within a specified city. We'll be exploring and analyzing the latitude and longitude values, including both, unique and duplicate values. The dataset is available and accessible to the public through the Simple maps website and can be downloaded at https://simplemaps.com/data/world-cities.

III. Environment Setup

The sorting process can be executed and accomplished using a variety of tools and programming languages, however we have opted to use 'Python Language' and 'Jupyter' framework to complete the tasks by using a fundamental mathematical concept. The infrastructure provided by Jupytor notebook allows to execute and view the code line by line, making it easier to monitor the algorithm sorting process at each step. Additionally, we chose to monitor the time and memory used during execution to support our decision-making process and ensure that our solutions are reliable and understandable.

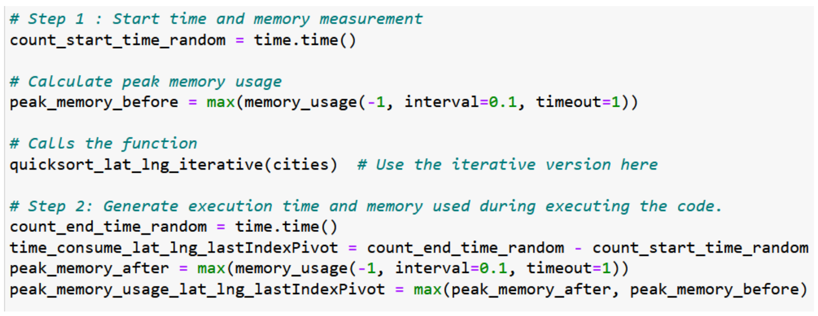


Work with pyhon version : 3.11.4

```
import sys
print('Work with Python version:', sys.version)
```

```
Work with Python version: 3.11.4 | packaged by Anaconda, Inc. | (main, Jul  5 2023, 13:38:37) [MSC v.1916 64 bit (AMD64)]
```

The objective of employing a sorting algorithm in this task extends beyond data organization. Load profiler is used to track the loading of the datasets and algorithm operations in order to keep track of the peak memory consumption and the execution time, to avoid memory overflowing with data and to study the algorithm's time consumption. Although tools such as profiers and



```
# Step 1 : Start time and memory measurement
count_start_time_random = time.time()

# Calculate peak memory usage
peak_memory_before = max(memory_usage(-1, interval=0.1, timeout=1))

# Calls the function
quicksort_lat_lng_iterative(cities)  # Use the iterative version here

# Step 2: Generate execution time and memory used during executing the code.
count_end_time_random = time.time()
time_consume_lat_lng_lastIndexPivot = count_end_time_random - count_start_time_random
peak_memory_after = max(memory_usage(-1, interval=0.1, timeout=1))
peak_memory_usage_lat_lng_lastIndexPivot = max(peak_memory_after, peak_memory_before)
```

timers may not provide exact accuracy regarding power usage, they can provide a clear picture of algorithm complexity (Aljabri et al., 2019). ** change to new screen short

IV. Methodology

1. Data Preprocessing

For the data preparation, we used data from the ' World Cities Database', which compiles the global city information. Although the dataset contains many columns with diverse data, we processed only three key columns: `city`, `lat`(latitude) and `lng`(longitude), which were relevant to us. First, we loaded the data. Thereafter, we extracted the required columns, latitude and longitude and removed any duplicates from the data to use only the unique values, as required. This step was crucial; hence the unnecessary data can impact the performance of merge sort, in terms of time consumption and memory usage. The final step consisted of data frame transformation into a one dimensional NumPy array for the latitude values and list of tuples combining `city`, `lat`(latitude) and `lng`(longitude).

2. Data Sorting

a. Merge sort

Merge sort algorithm is essentially a combination of divide and conquer sort that is extremely successful in breaking up unsorted arrays into smaller pieces and then reassembling them into a perfectly ordered collection. It works as the following:

1. <u>Setting up an unsorted array</u>: Contains the unsorted set of ‘size’ elements stored in the single block of contiguous storage.
2. <u>Dividing the array into two halves, left and right:</u> In case the division is not possible, one side of the dataset will contain one extra element than the other one.
3. <u>Recursive sorting:</u> The two divisions are sorted repeatedly using merge sort until the subdivision part has either one element or none. At this point, subdivision is regarded as sorted. This is because a presence of one element or a lack of an element are both considered as sorted.
4. <u>Conquer:</u> Merging the two divisions into a single sorted set.
5. <u>Merging larger parts:</u> Combining these sorted sequences into larger sorted sequences to continue with the merging process.
6. <u>Final Merge:</u> Performs the final merge, combining the steps 4 and 5 into a fully sorted dataset.

b) Quick sort

Quick Sort is quite similar to the merge sort algorithm, is it an efficient method to sort, where the amount of data increases without it proportionally increasing the process time, making it extremely useful to handle large time complex data. There are key differences that sets them apart.

a) Array Division:

- In quick sort a pivot element is used to split the array into two sections where the one side has the elements that are less than the pivot and other side that consists of elements that are greater than the pivot. Thereafter the pivot is positioned in its final sorted position.

a) Memory Usage:

- Quick sort is frequently implemented in-place meaning that it does not require additional memory compared to the size of the input.

b) Stability:

- When sorting it is possible that the order of the equal elements will change hence it is not stable.

c) Time Complexity:

- Time complexity is defined by the recurrence relation T(n) = 2T(n/2) + O(n), where the outcome is O(nLogn) and it exhibits the same time complexity across worst, average, and best cases.

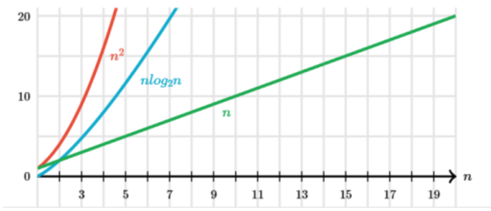
d) Pivot Selection:

- The choice of the pivot element has a big impact on efficiency hence a good pivot will help balancing the partitions and will contribute with optimal performance.

C) Time and Complexity

Overall time and complexity of the both:

Algorithm	Worst case running time	Best-case running time	Average case running time
Merge Sort	O(n*log n)	O(n*log n)	O(n*log n)
Quick sort	$O(n^2)$	O(n*log n)	O(n*log n)



Images are from : Khan Academy. (n.d.). *Challenge: Implement quicksort | Quick sort | Algorithms | Computer science theory | Computing*. [online] Available at: https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/pc/challenge-implement-quicksort

- **Merge sort Time Complexity & Memory**

Merge sort is an efficient sorting technique, where the increasing data amounts do not cause an equivalent increase in the processing time, making it extremely helpful when handling large amounts of time-complex data. Algorithm complexity works as the following:

1) Time complexity

The recurrence relation T(n) = 2T(n/2) + O(n) represents the time complexity of merge sort. The array is always divided in half by the merge sort, therefore the three cases (worst, average and best) are all the same. Each division cuts the size of the array in half until the array gets down to one single element in the array’s size relation. At this point the log n part of time complexity O(n log n) is applied. During the merging process, merge sort employs the linear time to combine the two parts, which means that all elements are combined simultaneously for each pair of subarrays being merged. Therefore, to merge them in the size of n, n operations will be required. The merging happens for each level of division and also log(n) levels, the total time complexity becomes O(n log n) (Gautam, n.d).

2) Memory:

O(n), sorting in place, does not employ a divide and conquer algorithm. It works by creating a new array of lists that holds the combined subsets of data throughout the sorting process. Merging sort does not affect the algorithm's overall memory efficiency since, after it is finished, the extra space can be used for the subsequent merging. However, even if merge sort does use more memory, the quantity is not increasing with each merge.

- **Quick Sort Time Complexity & Memory:**

1) Time Complexity:

- Linear Time (O(n)): The runtime grows in direct proportion to the size of the input or the number of elements.
- Constant Time (O(1)): When the runtime is not changed as the size increases it is represented with a straight line.
- Quadratic Time (O(n^2)): The runtime increases correspondingly to the square of the input size.

2) Memory:

- Quick sort eliminates the need for the extra memory hence it operates in place, but in some cases, there might be a need for extra memory for optimizations.

The execution time is calculated by the formula T = an + b, equals to O(N) where the O(n) represents the time complexity. T = cn^2 + dn + e would be the outcome when finding the fastest-growing term and eliminating the coefficient.

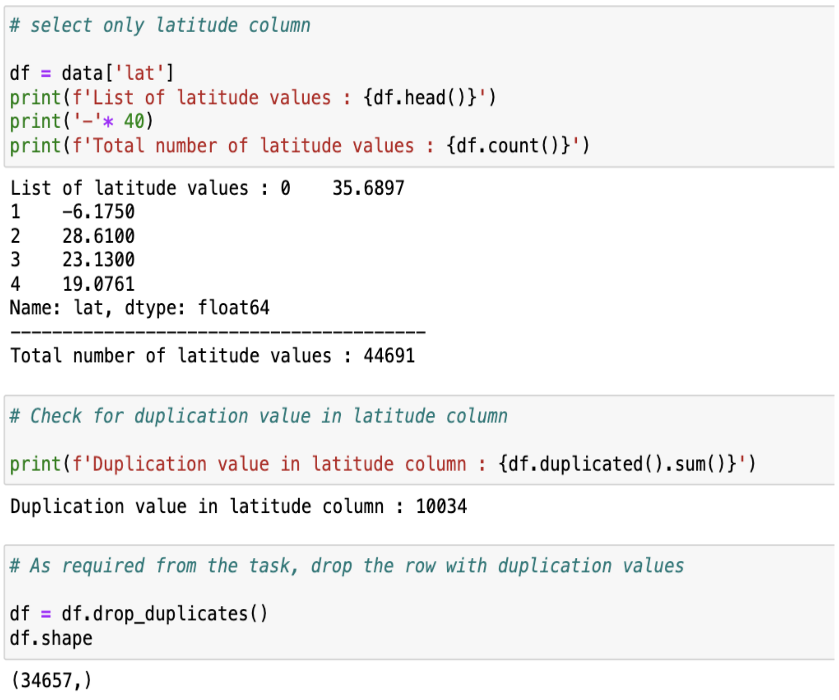
V. Results and Discussion

1) Merge Sort Problem:

a) Implement a proper merge sort algorithm so that all city latitudes are in an ordered list. We'll use the unique latitude values of each city only.

1) Handling Duplicate Values

Initially, the latitude values column consists of 44,691 rows, where 10,034 rows are duplicate values. As requested from the task, we dropped the duplicated values and 34,657 rows of unique values. Mergesort works by sorting arrays or lists and the original data type is 'Pandas dataFrame' Thereafter the data was ready to convert into a one dimensional 'numpy array'.



```
# select only latitude column

df = data['lat']
print(f'List of latitude values : {df.head()}')
print('-'* 40)
print(f'Total number of latitude values : {df.count()}')
```

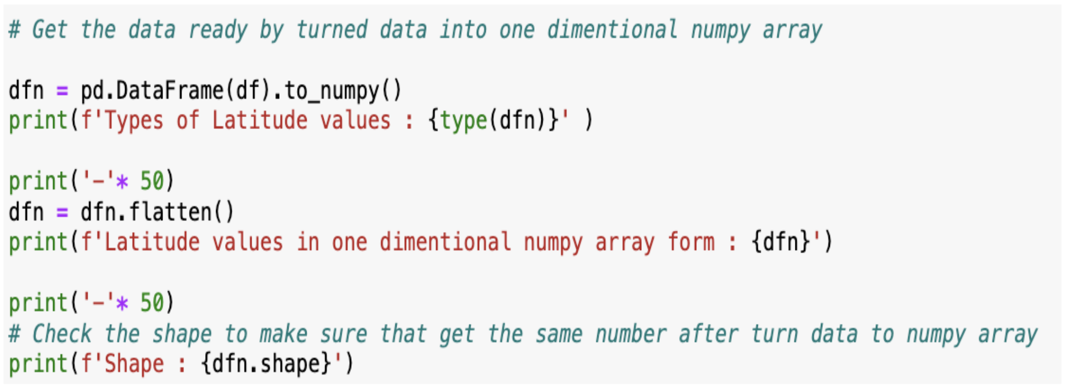
```
List of latitude values : 0    35.6897
1    -6.1750
2    28.6100
3    23.1300
4    19.0761
Name: lat, dtype: float64
----------------------------------------
Total number of latitude values : 44691
```

```
# Check for duplication value in latitude column

print(f'Duplication value in latitude column : {df.duplicated().sum()}')
```

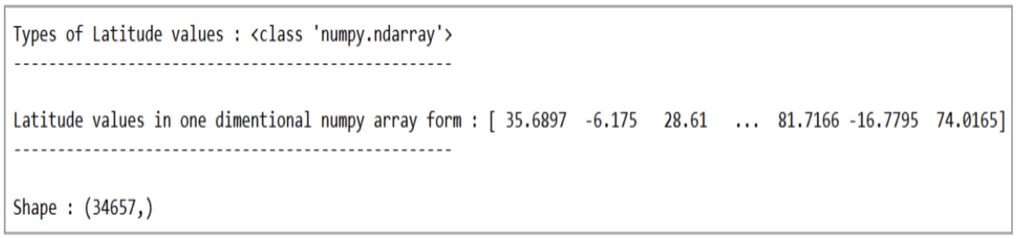
```
Duplication value in latitude column : 10034
```

```
# As required from the task, drop the row with duplication values

df = df.drop_duplicates()
df.shape
```

```
(34657,)
```



```
# Get the data ready by turned data into one dimentional numpy array

dfn = pd.DataFrame(df).to_numpy()
print(f'Types of Latitude values : {type(dfn)}' )

print('-'* 50)
dfn = dfn.flatten()
print(f'Latitude values in one dimentional numpy array form : {dfn}')

print('-'* 50)
# Check the shape to make sure that get the same number after turn data to numpy array
print(f'Shape : {dfn.shape}')
```



```
Types of Latitude values : <class 'numpy.ndarray'>
--------------------------------------------------

Latitude values in one dimentional numpy array form : [ 35.6897  -6.175   28.61   ...  81.7166 -16.7795  74.0165]
--------------------------------------------------

Shape : (34657,)
```

Figure 1: Turn data into Numpy array

2) Merge Sort Algorithm Implementation to order latitude values in all the cities.

Figure 2 which aligns with Figure 3 (Pseudocode**)** showcases how the merge sort algorithm is applied to sort city latitudes in an ordered list.

1. Divide and recursive:

The array is partitioned by n keys in a non-decreasing sequence into two subarrays, each containing n/2 items. Afterwards, Recursive(sort) sorts each subarray. Unless the array is sufficiently small, recursively apply merge sort for further division and sort it.

2. Merge (combine):

Creates a single sorted array by combining the sorted subarrays. In this step, the non-decreasing sequence is preserved by combining the subarray's items in the correct order.

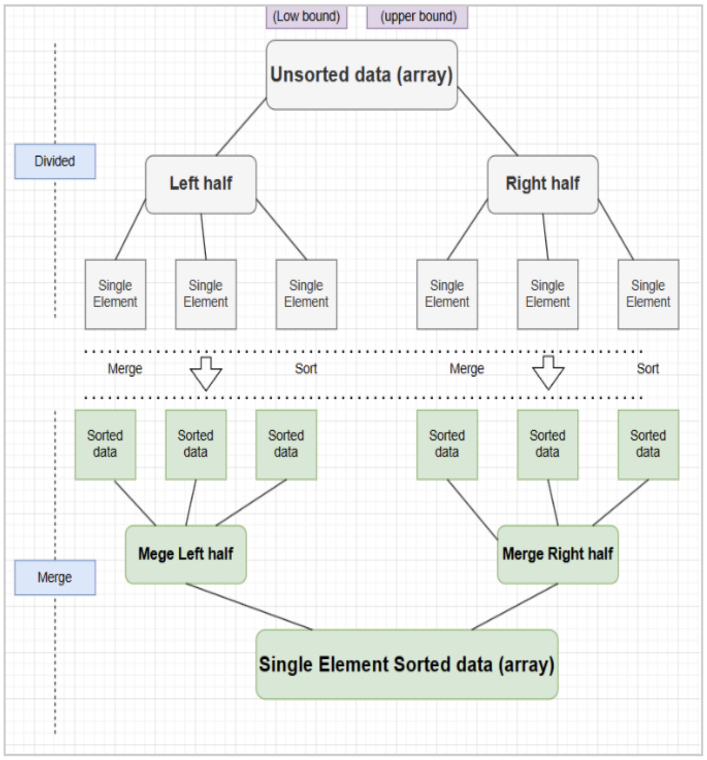
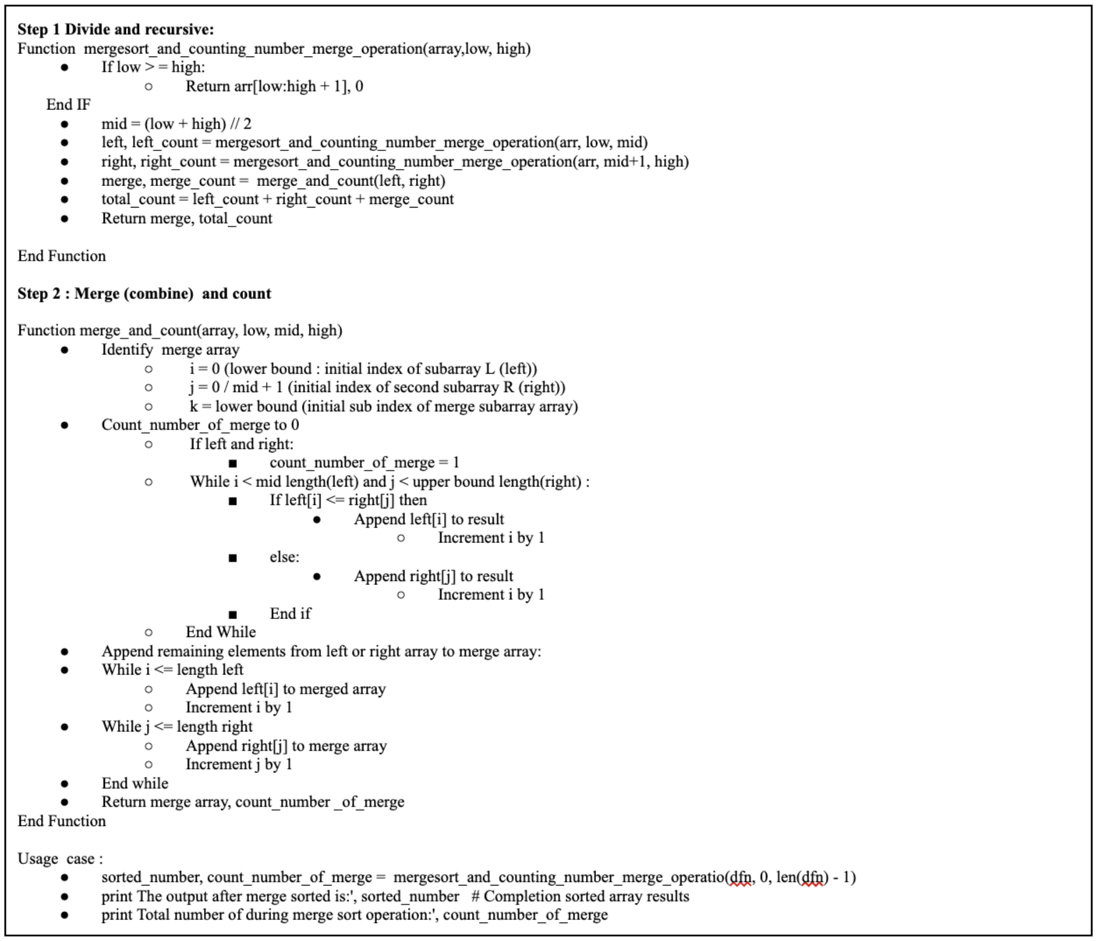


Figure 2 Pipeline for merge sort

The Pseudocode in the figure 3 represents how the merge sort has been used to solve the problem. Additionally, it also counts the number of merge operations made while sorting the array. This helps with the understanding of the inside code implementation.

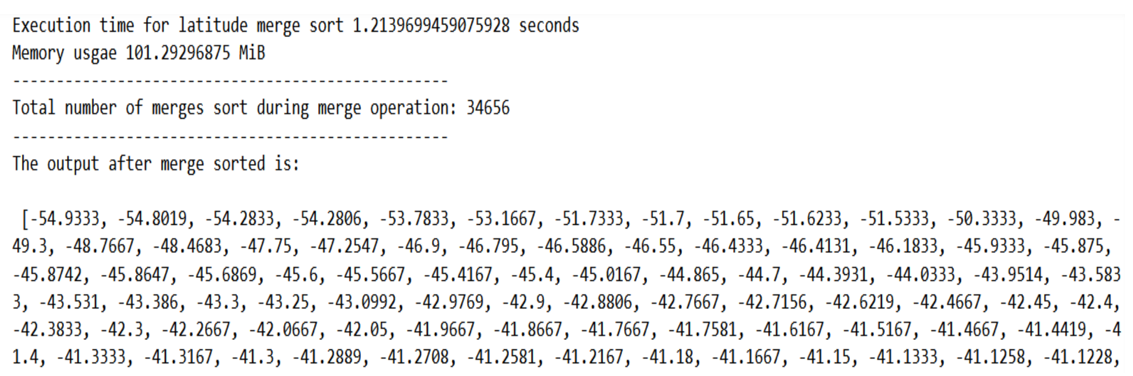
Pseudocode



```
Step 1 Divide and recursive:
Function  mergesort_and_counting_number_merge_operation(array,low, high)
    ●   If low > = high:
            ○   Return arr[low:high + 1], 0
    End IF
    ●   mid = (low + high) // 2
    ●   left, left_count = mergesort_and_counting_number_merge_operation(arr, low, mid)
    ●   right, right_count = mergesort_and_counting_number_merge_operation(arr, mid+1, high)
    ●   merge, merge_count =  merge_and_count(left, right)
    ●   total_count = left_count + right_count + merge_count
    ●   Return merge, total_count

End Function

Step 2 : Merge (combine)  and count

Function merge_and_count(array, low, mid, high)
    ●   Identify  merge array
            ○   i = 0 (lower bound : initial index of subarray L (left))
            ○   j = 0 / mid + 1 (initial index of second subarray R (right))
            ○   k = lower bound (initial sub index of merge subarray array)
    ●   Count_number_of_merge to 0
            ○   If left and right:
                    ■   count_number_of_merge = 1
            ○   While i < mid length(left) and j < upper bound length(right) :
                    ■   If left[i] <= right[j] then
                            ●   Append left[i] to result
                                    ○   Increment i by 1
                    ■   else:
                            ●   Append right[j] to result
                                    ○   Increment i by 1
                    ■   End if
            ○   End While
    ●   Append remaining elements from left or right array to merge array:
    ●   While i <= length left
            ○   Append left[i] to merged array
            ○   Increment i by 1
    ●   While j <= length right
            ○   Append right[j] to merge array
            ○   Increment j by 1
    ●   End while
    ●   Return merge array, count_number _of_merge
End Function

Usage  case :
    ●   sorted_number, count_number_of_merge =  mergesort_and_counting_number_merge_operatio(dfn, 0, len(dfn) - 1)
    ●   print The output after merge sorted is:', sorted_number   # Completion sorted array results
    ●   print Total number of during merge sort operation:', count_number_of_merge
```

Figure 3 Pseudocode

The output of the code showcases that the total number of merge sort during merge operation is 34656. Additionally, the output of the sorted merge showcasing all the city latitudes in the ordered list can also be seen in the image below.

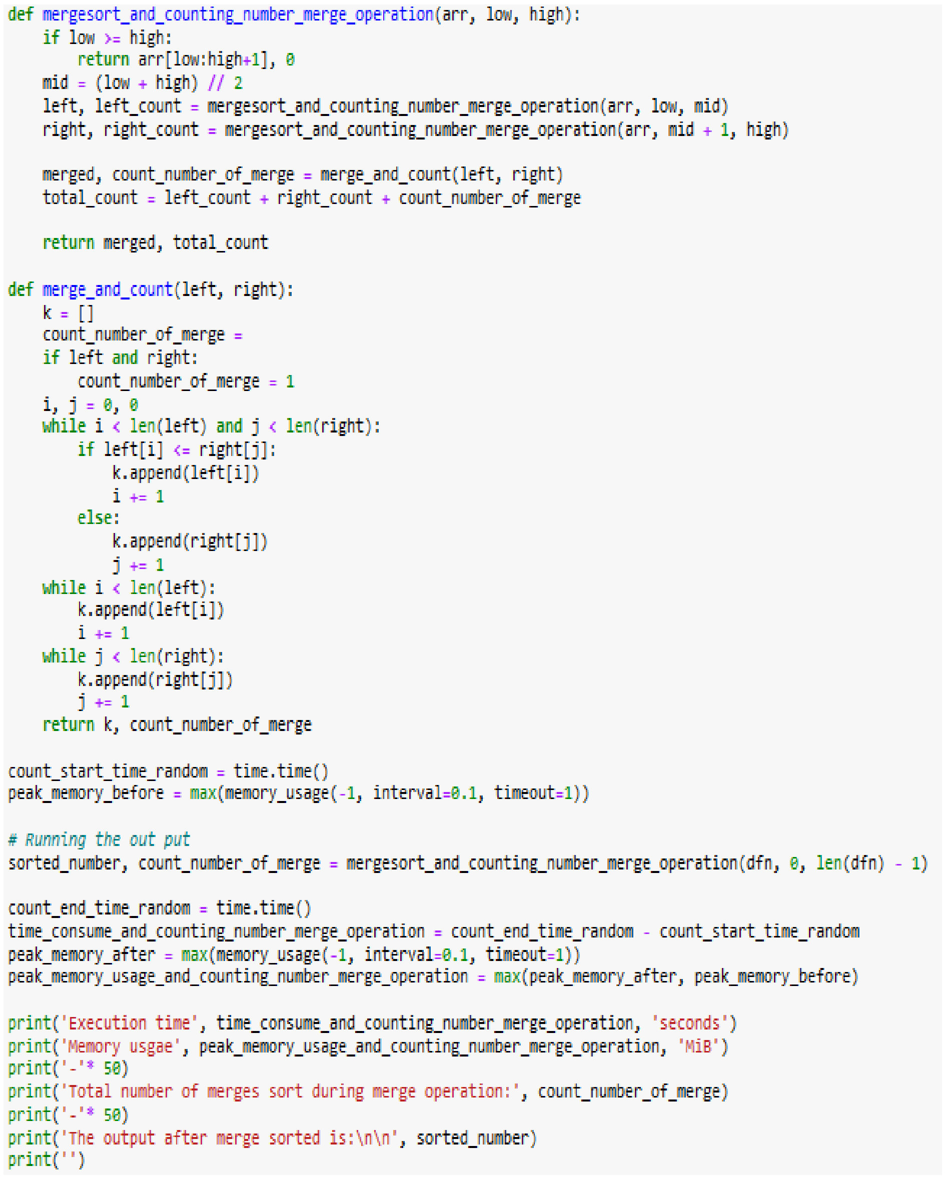


```
Execution time for latitude merge sort 1.2139699459075928 seconds
Memory usgae 101.29296875 MiB
-------------------------------------------------
Total number of merges sort during merge operation: 34656
-------------------------------------------------
The output after merge sorted is:

 [-54.9333, -54.8019, -54.2833, -54.2806, -53.7833, -53.1667, -51.7333, -51.7, -51.65, -51.6233, -51.5333, -50.3333, -49.983, -
49.3, -48.7667, -48.4683, -47.75, -47.2547, -46.9, -46.795, -46.5886, -46.55, -46.4333, -46.4131, -46.1833, -45.9333, -45.875,
-45.8742, -45.8647, -45.6869, -45.6, -45.5667, -45.4167, -45.4, -45.0167, -44.865, -44.7, -44.3931, -44.0333, -43.9514, -43.583
3, -43.531, -43.386, -43.3, -43.25, -43.0992, -42.9769, -42.9, -42.8806, -42.7667, -42.7156, -42.6219, -42.4667, -42.45, -42.4,
-42.3833, -42.3, -42.2667, -42.0667, -42.05, -41.9667, -41.8667, -41.7667, -41.7581, -41.6167, -41.5167, -41.4667, -41.4419, -4
1.4, -41.3333, -41.3167, -41.3, -41.2889, -41.2708, -41.2581, -41.2167, -41.18, -41.1667, -41.15, -41.1333, -41.1258, -41.1228,
```

Analysis of the result from 1a) : Merge sort to latitude data sorting

a) Merging process

(The code presented in the image prints the answer showcased in the image above).



```python
def mergesort_and_counting_number_merge_operation(arr, low, high):
    if low >= high:
        return arr[low:high+1], 0
    mid = (low + high) // 2
    left, left_count = mergesort_and_counting_number_merge_operation(arr, low, mid)
    right, right_count = mergesort_and_counting_number_merge_operation(arr, mid + 1, high)

    merged, count_number_of_merge = merge_and_count(left, right)
    total_count = left_count + right_count + count_number_of_merge

    return merged, total_count

def merge_and_count(left, right):
    k = []
    count_number_of_merge =
    if left and right:
        count_number_of_merge = 1
    i, j = 0, 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            k.append(left[i])
            i += 1
        else:
            k.append(right[j])
            j += 1
    while i < len(left):
        k.append(left[i])
        i += 1
    while j < len(right):
        k.append(right[j])
        j += 1
    return k, count_number_of_merge

count_start_time_random = time.time()
peak_memory_before = max(memory_usage(-1, interval=0.1, timeout=1))

# Running the out put
sorted_number, count_number_of_merge = mergesort_and_counting_number_merge_operation(dfn, 0, len(dfn) - 1)

count_end_time_random = time.time()
time_consume_and_counting_number_merge_operation = count_end_time_random - count_start_time_random
peak_memory_after = max(memory_usage(-1, interval=0.1, timeout=1))
peak_memory_usage_and_counting_number_merge_operation = max(peak_memory_after, peak_memory_before)

print('Execution time', time_consume_and_counting_number_merge_operation, 'seconds')
print('Memory usgae', peak_memory_usage_and_counting_number_merge_operation, 'MiB')
print('-'* 50)
print('Total number of merges sort during merge operation:', count_number_of_merge)
print('-'* 50)
print('The output after merge sorted is:\n\n', sorted_number)
print('')
```

Step 1: Divide and Recursive: The objective is to sort the points in a 34,657-element array into an ordered list. This transforms the sorting problem from an array (arr[0...n-1]) to sorting a subarray (arr[i..j]).

- The array is initially divided into two halves, with lower and upper bounds set to 0 and 34,657.
- Dividing the sum of bounds by two gives us approximately 17,273.5.
- The algorithm sorts the first half of the array from index 0 to 17,274 and thereafter the second half approximately index 17,275 to 34,657.

Step 2 : Merge:

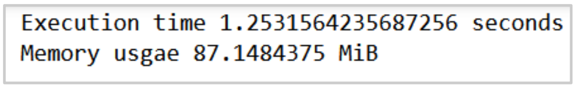
Here, a larger sorted array is created by merging the two sorted subarrays. The smallest unmerged element is compared and is added to a new array while preserving order. Thereafter, a recursive process is varied out for every separated part until a single sorted sequence is achieved.

b) Performance Evaluation: Time complexity & Memory Usage

The efficiency of merge sort is evaluated by its time complexity O(n log n) for an array of ‘n’ elements, whose running time is better than the other type of sorting :

1. Time complexity: The division process, paired with linear-time merging, enhances overall algorithm efficiency.

2. Memory usage: Even though merge sort requires additional room to merge, the impact on memory efficiency is minimized as the space is reused for subsequent merges.



```
Execution time 1.2531564235687256 seconds
Memory usgae 87.1484375 MiB
```

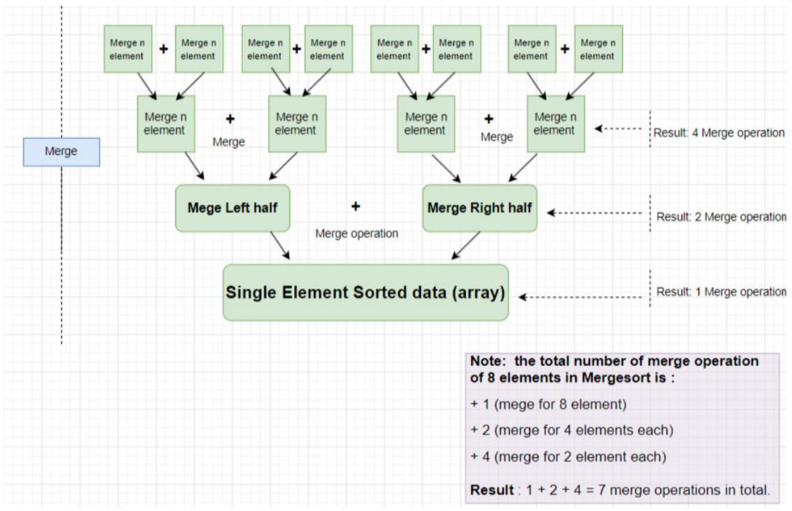
In conclusion, analysis validates that merge sort is a suitable method for arranging a dataset of latitudes that contains 34,657 components. The efficiency is demonstrated in the computing stage with the measured peak memory and execution time falling within acceptable bounds for the given data size.

b) Count the number of merges needed to sort the dataset. Does it change if you randomly order the list before sorting? Why/why not?

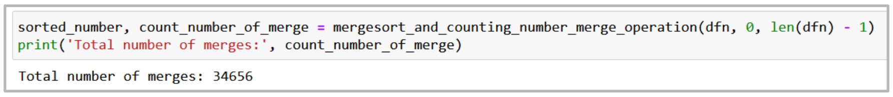
The task requested to determine the count of merges required for dataset sorting. There are two possible interpretations for the phrase "count the number of merges”: Counting the comparisons made in each element during the merge sort process or counting the total number of times the subarray is combined. Hence the task was unclear, we opted to approach both scenarios:

1) Count the number of merges during operation process

Hence, we are working with a large dataset consisting of 34,657 components, The Figure 4 provides an easy example using a small subset of 8 elements to explain how the count number of operations work in a smaller number of elements. As the details in step 2 of pseudocode, “merge to sort data” We start with an unsorted array that contains (8) elements and a total of 7 merge operations are required to sort the array. Finally splitting every subarray in half and counting the merges that bring them back together, as seen in Figure 4 1 (whole element of array 8) + 2(two of 4 sorted elements) + 4(four of 2 sorted elements) for 8 items), total of 7 merge operations.


Merge n element
+
Merge
Mege Left half
Merge Right half
Merge operation
Single Element Sorted data (array)
Result: 4 Merge operation
Result: 2 Merge operation
Result: 1 Merge operation
Note: the total number of merge operation of 8 elements in Mergesort is :
+ 1 (mege for 8 element)
+ 2 (merge for 4 elements each)
+ 4 (merge for 2 element each)
Result : 1 + 2 + 4 = 7 merge operations in total.

After implementing the merge sort algorithm in part ‘a’ to organize city latitudes, which involves counting the merges needed to sort the dataset. A merge operation is applied to each occurrence, where two single-element arrays are combined to form one. The number of merge operations for an array of ‘n’ elements is typically n-1 as the merge sort naturally occurs, shown by ‘len(dfn) – 1.



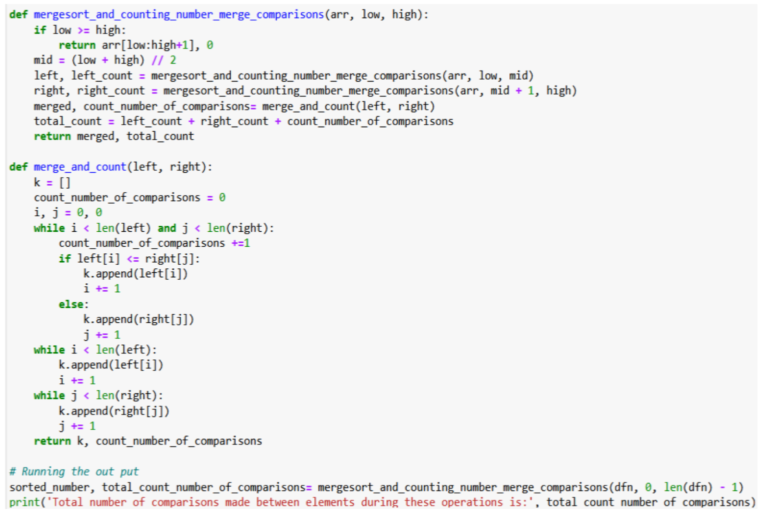
```
sorted_number, count_number_of_merge = mergesort_and_counting_number_merge_operation(dfn, 0, len(dfn) - 1)
print('Total number of merges:', count_number_of_merge)
```

```
Total number of merges: 34656
```

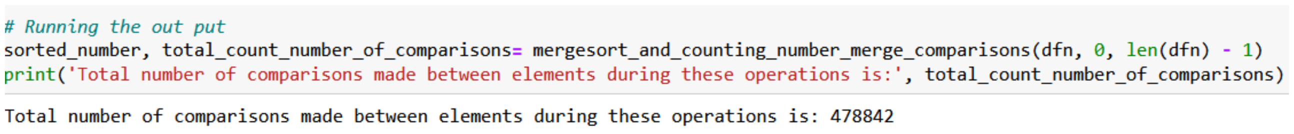
In part ‘a’, we count the merge operations while ordering latitudes. The ‘count_number_of_merge’ variable in the ‘merge’ function, set to ‘1’ for each call of 'mergesort_and_counting_number_merge_operation,' keeps track of the total number of merge operations completed during the sorting procedure. The count of merge operations is 34656, as shown in the image above.

2) Count the number of comparisons

Mergesort_and_counting_number _merge_comparisonsmethod was used to count the number of comparisons made during the sorting process.An input array (arr) is quickly sorted by repeatedly dividing it into smaller subarrays, sorting the resulting union, and counting the number of comparisons made between elements during the operations. As shown in the image, the total number of comparisons is 47 8842.



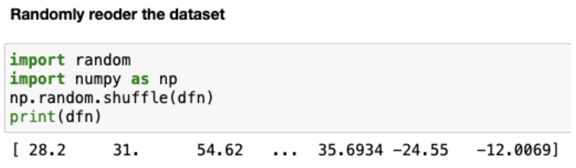
```python
def mergesort_and_counting_number_merge_comparisons(arr, low, high):
    if low >= high:
        return arr[low:high+1], 0
    mid = (low + high) // 2
    left, left_count = mergesort_and_counting_number_merge_comparisons(arr, low, mid)
    right, right_count = mergesort_and_counting_number_merge_comparisons(arr, mid + 1, high)
    merged, count_number_of_comparisons= merge_and_count(left, right)
    total_count = left_count + right_count + count_number_of_comparisons
    return merged, total_count

def merge_and_count(left, right):
    k = []
    count_number_of_comparisons = 0
    i, j = 0, 0
    while i < len(left) and j < len(right):
        count_number_of_comparisons +=1
        if left[i] <= right[j]:
            k.append(left[i])
            i += 1
        else:
            k.append(right[j])
            j += 1
    while i < len(left):
        k.append(left[i])
        i += 1
    while j < len(right):
        k.append(right[j])
        j += 1
    return k, count_number_of_comparisons

# Running the out put
sorted_number, total_count_number_of_comparisons= mergesort_and_counting_number_merge_comparisons(dfn, 0, len(dfn) - 1)
print('Total number of comparisons made between elements during these operations is:', total count number of comparisons)
```



```
# Running the out put
sorted_number, total_count_number_of_comparisons= mergesort_and_counting_number_merge_comparisons(dfn, 0, len(dfn) - 1)
print('Total number of comparisons made between elements during these operations is:', total_count_number_of_comparisons)

Total number of comparisons made between elements during these operations is: 478842
```

B.1) Does it change if you randomly order the list before sorting? Why/why not ?
Effect of randomly ordering the List before sorting.

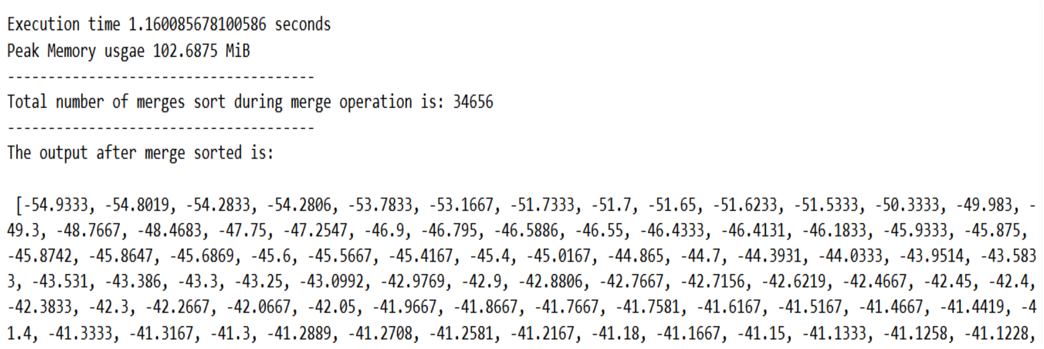
In this section, the question asks whether the dataset's random order influences the number of merges needed for the merge sort method. In order to figure this out, we will write a code that randomly rearranges the dataset and then apply merge sort to determine if the number of merge changes when sorting the array of latitude numbers.



Randomly reoder the dataset

```
import random
import numpy as np
np.random.shuffle(dfn)
print(dfn)

[ 28.2      31.       54.62    ...  35.6934 -24.55     -12.0069]
```

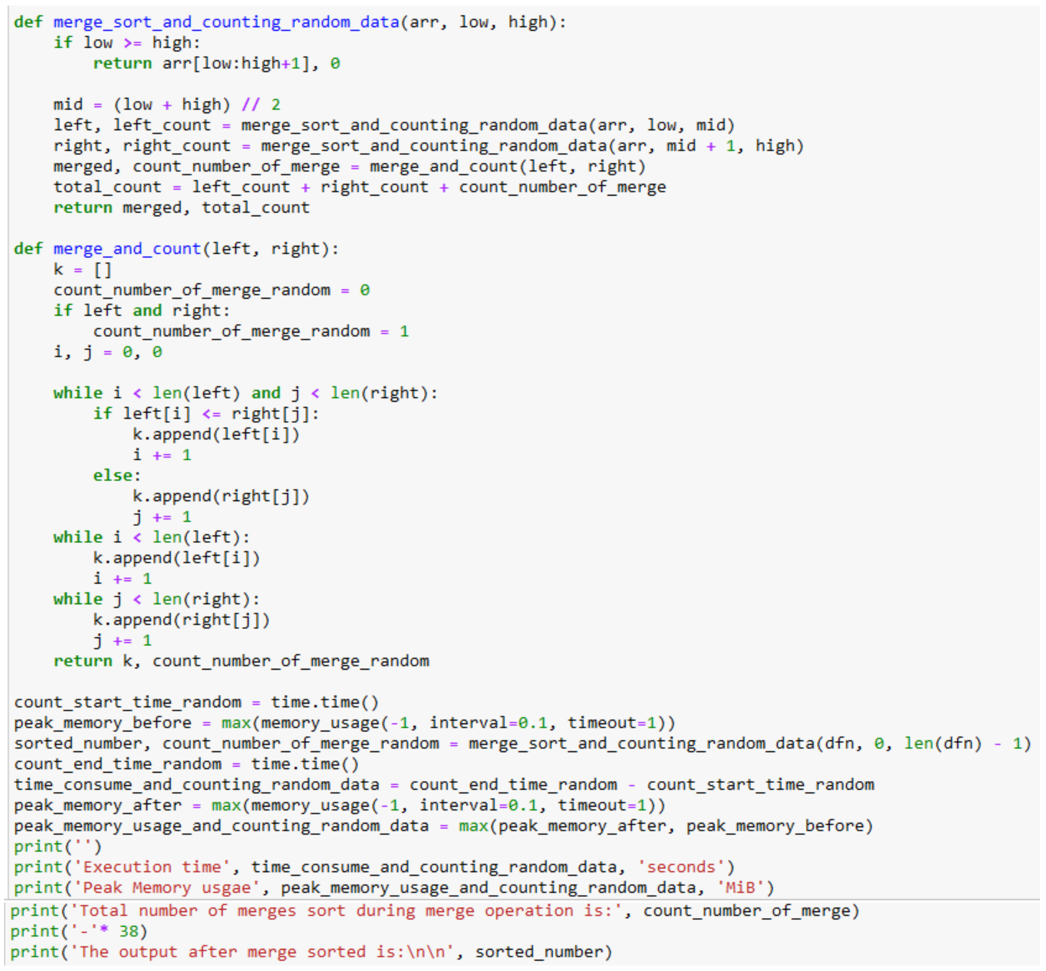
Analyzing whether it changes in both of our approached scenarios:



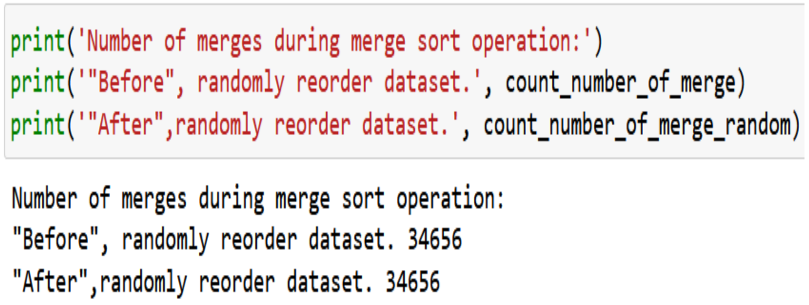
```
Execution time 1.160085678100586 seconds
Peak Memory usgae 102.6875 MiB
--------------------------------------
Total number of merges sort during merge operation is: 34656
--------------------------------------
The output after merge sorted is:

 [-54.9333, -54.8019, -54.2833, -54.2806, -53.7833, -53.1667, -51.7333, -51.7, -51.65, -51.6233, -51.5333, -50.3333, -49.983, -
49.3, -48.7667, -48.4683, -47.75, -47.2547, -46.9, -46.795, -46.5886, -46.55, -46.4333, -46.4131, -46.1833, -45.9333, -45.875,
-45.8742, -45.8647, -45.6869, -45.6, -45.5667, -45.4167, -45.4, -45.0167, -44.865, -44.7, -44.3931, -44.0333, -43.9514, -43.583
3, -43.531, -43.386, -43.3, -43.25, -43.0992, -42.9769, -42.9, -42.8806, -42.7667, -42.7156, -42.6219, -42.4667, -42.45, -42.4,
-42.3833, -42.3, -42.2667, -42.0667, -42.05, -41.9667, -41.8667, -41.7667, -41.7581, -41.6167, -41.5167, -41.4667, -41.4419, -4
1.4, -41.3333, -41.3167, -41.3, -41.2889, -41.2708, -41.2581, -41.2167, -41.18, -41.1667, -41.15, -41.1333, -41.1258, -41.1228,
```

1) Counting the number of merges during merge sort operation.

The code shown in the image was used to calculate the total number of merge operations produced, which offers information on the merge sort algorithm’s efficiency. Additionally, the code also measures and reports the memory usage, the sorted array following the merge sort operation on a randomly ordered latitude dataset (dfn), and the execution time. Helping to analyze how well it performs in terms of both time and space complexity.



```python
def merge_sort_and_counting_random_data(arr, low, high):
    if low >= high:
        return arr[low:high+1], 0

    mid = (low + high) // 2
    left, left_count = merge_sort_and_counting_random_data(arr, low, mid)
    right, right_count = merge_sort_and_counting_random_data(arr, mid + 1, high)
    merged, count_number_of_merge = merge_and_count(left, right)
    total_count = left_count + right_count + count_number_of_merge
    return merged, total_count

def merge_and_count(left, right):
    k = []
    count_number_of_merge_random = 0
    if left and right:
        count_number_of_merge_random = 1
    i, j = 0, 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            k.append(left[i])
            i += 1
        else:
            k.append(right[j])
            j += 1
    while i < len(left):
        k.append(left[i])
        i += 1
    while j < len(right):
        k.append(right[j])
        j += 1
    return k, count_number_of_merge_random

count_start_time_random = time.time()
peak_memory_before = max(memory_usage(-1, interval=0.1, timeout=1))
sorted_number, count_number_of_merge_random = merge_sort_and_counting_random_data(dfn, 0, len(dfn) - 1)
count_end_time_random = time.time()
time_consume_and_counting_random_data = count_end_time_random - count_start_time_random
peak_memory_after = max(memory_usage(-1, interval=0.1, timeout=1))
peak_memory_usage_and_counting_random_data = max(peak_memory_after, peak_memory_before)
print('')
print('Execution time', time_consume_and_counting_random_data, 'seconds')
print('Peak Memory usgae', peak_memory_usage_and_counting_random_data, 'MiB')
print('Total number of merges sort during merge operation is:', count_number_of_merge)
print('-'* 38)
print('The output after merge sorted is:\n\n', sorted_number)
```



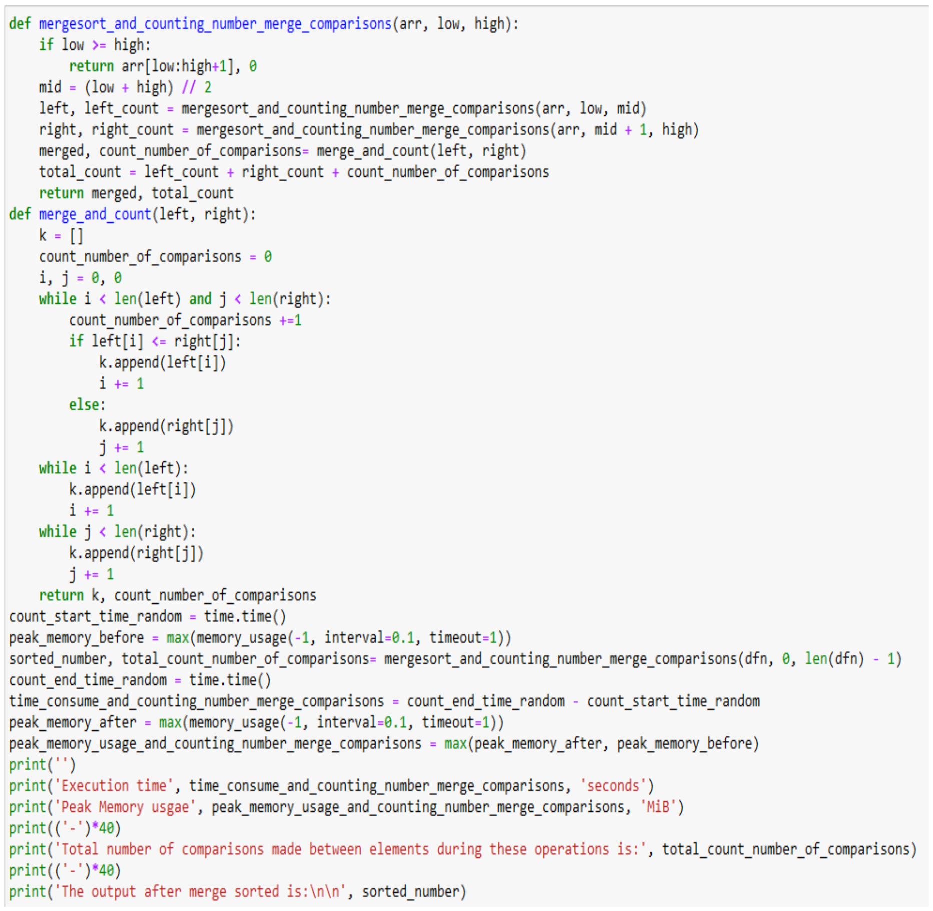
```python
print('Number of merges during merge sort operation:')
print('"Before", randomly reorder dataset.', count_number_of_merge)
print('"After",randomly reorder dataset.', count_number_of_merge_random)
```

```
Number of merges during merge sort operation:
"Before", randomly reorder dataset. 34656
"After",randomly reorder dataset. 34656
```

In conclusion, the **total number of merges does not change after randomly reordering the data** during the merge sort operation. This because, the number of splits and merges depend on the total number of elements rather on their initial order. Therefor regardless of whether the list is randomly ordered or not, the number of merges required remains the same. Additionally, randomly ordered latitude dataset doesn’t significantly impact time and memory usage, hence the time execution and memory usage remains in the same range when compared to (a).

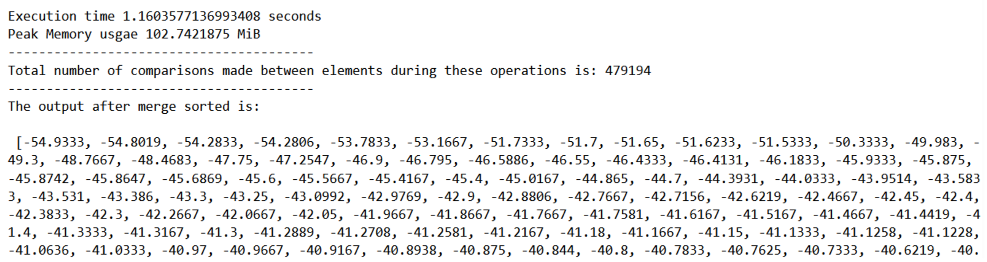
II. Counting the number of comparisons

Used the code shown in the image to print the total number of comparisons needed to completely sort a dataset using the MergeSort algorithm.



```
def mergesort_and_counting_number_merge_comparisons(arr, low, high):
    if low >= high:
        return arr[low:high+1], 0
    mid = (low + high) // 2
    left, left_count = mergesort_and_counting_number_merge_comparisons(arr, low, mid)
    right, right_count = mergesort_and_counting_number_merge_comparisons(arr, mid + 1, high)
    merged, count_number_of_comparisons= merge_and_count(left, right)
    total_count = left_count + right_count + count_number_of_comparisons
    return merged, total_count
def merge_and_count(left, right):
    k = []
    count_number_of_comparisons = 0
    i, j = 0, 0
    while i < len(left) and j < len(right):
        count_number_of_comparisons +=1
        if left[i] <= right[j]:
            k.append(left[i])
            i += 1
        else:
            k.append(right[j])
            j += 1
    while i < len(left):
        k.append(left[i])
        i += 1
    while j < len(right):
        k.append(right[j])
        j += 1
    return k, count_number_of_comparisons
count_start_time_random = time.time()
peak_memory_before = max(memory_usage(-1, interval=0.1, timeout=1))
sorted_number, total_count_number_of_comparisons= mergesort_and_counting_number_merge_comparisons(dfn, 0, len(dfn) - 1)
count_end_time_random = time.time()
time_consume_and_counting_number_merge_comparisons = count_end_time_random - count_start_time_random
peak_memory_after = max(memory_usage(-1, interval=0.1, timeout=1))
peak_memory_usage_and_counting_number_merge_comparisons = max(peak_memory_after, peak_memory_before)
print('')
print('Execution time', time_consume_and_counting_number_merge_comparisons, 'seconds')
print('Peak Memory usgae', peak_memory_usage_and_counting_number_merge_comparisons, 'MiB')
print(('-')*40)
print('Total number of comparisons made between elements during these operations is:', total_count_number_of_comparisons)
print(('-')*40)
print('The output after merge sorted is:\n\n', sorted_number)
```

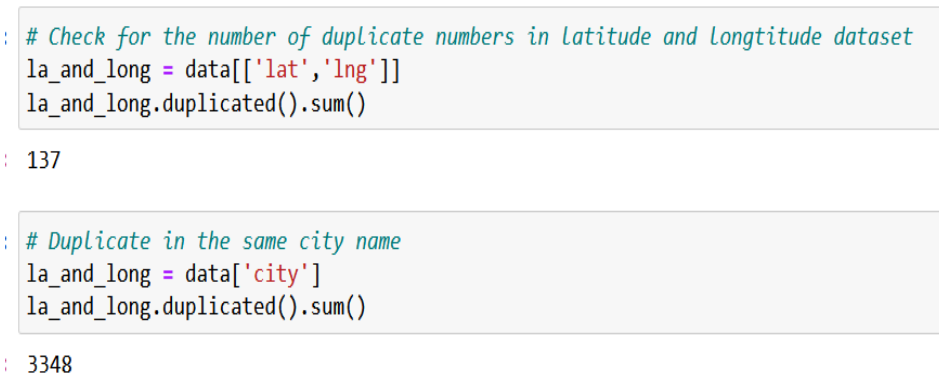
In conclusion, the **<u>total number of comparisons does change</u>** when we randomly order the list before sorting while counting the number of comparisons. The explanation for this is that the comparisons are individual evaluations made to determine the hierarchy of components in two subarrays, meaning there may be a difference in the number of comparisons depending on the array's original order . Less comparisons might be needed when dealing with a pre-sorted array, which would improve the performance of the merge stage but if the array is arranged randomly, additional comparisons may be needed to establish the order of the subarrays, hence the order of an element provides no guarantee of relative ordering between the subarrays, necessitating comparisons to determine their order in contrast to the count of merges. That is dependent on the size of the dataset rather than its original order meaning that the number of merge operations is independent of whether the array is sorted or arranged at random. Moreover, we observed that execution time and peak memory usage are comparable to the one recorded in part (a) and (b), meaning that the algorithm's time complexity and memory usage are consistent across different runs, even when performed in different order lists.



```
Execution time 1.1603577136993408 seconds
Peak Memory usgae 102.7421875 MiB
----------------------------------------
Total number of comparisons made between elements during these operations is: 479194
----------------------------------------
The output after merge sorted is:

 [-54.9333, -54.8019, -54.2833, -54.2806, -53.7833, -53.1667, -51.7333, -51.7, -51.65, -51.6233, -51.5333, -50.3333, -49.983, -
49.3, -48.7667, -48.4683, -47.75, -47.2547, -46.9, -46.795, -46.5886, -46.55, -46.4333, -46.4131, -46.1833, -45.9333, -45.875,
-45.8742, -45.8647, -45.6869, -45.6, -45.5667, -45.4167, -45.4, -45.0167, -44.865, -44.7, -44.3931, -44.0333, -43.9514, -43.583
3, -43.531, -43.386, -43.3, -43.25, -43.0992, -42.9769, -42.9, -42.8806, -42.7667, -42.7156, -42.6219, -42.4667, -42.45, -42.4,
-42.3833, -42.3, -42.2667, -42.0667, -42.05, -41.9667, -41.8667, -41.7667, -41.7581, -41.6167, -41.5167, -41.4667, -41.4419, -4
1.4, -41.3333, -41.3167, -41.3, -41.2889, -41.2708, -41.2581, -41.2167, -41.18, -41.1667, -41.15, -41.1333, -41.1258, -41.1228,
-41.0636, -41.0333, -40.97, -40.9667, -40.9167, -40.8938, -40.875, -40.844, -40.8, -40.7833, -40.7625, -40.7333, -40.6219, -40.
```

C) Implement a proper merge sort algorithm so that the (latitude, and longitude) pairs are in an ordered list. What distance measure is used? Use the latitude and longitude values for each city.

Decision Making: Consider duplicates in city names, latitude, and longitude pairs. Since the question doesn’t require identifying unique values, we choose to sort all latitude and longitude pairs, including duplicate city names. Another reason is that a large number of cities with latitude and longitude duplicate values that are less than a degree apart may exist. By doing this, the entire dataset is organized based on geographic coordinates without removing duplicates. To support this decision, the analysis reveals 3348 duplicate city names and 137 instances of duplication in latitude and longitude. To sort by pairs of latitude and longitude with 44,691 pairs of geographical coordinates in the data, we decided to employ two methods to order the list. We choose primarily to sort by latitude and secondarily by longitude, ensuring that the dataset is ordered according to meaningful geographical criteria using a merge sort algorithm, handling the large dataset efficiently.



```
# Check for the number of duplicate numbers in latitude and longtitude dataset
la_and_long = data[['lat','lng']]
la_and_long.duplicated().sum()
```

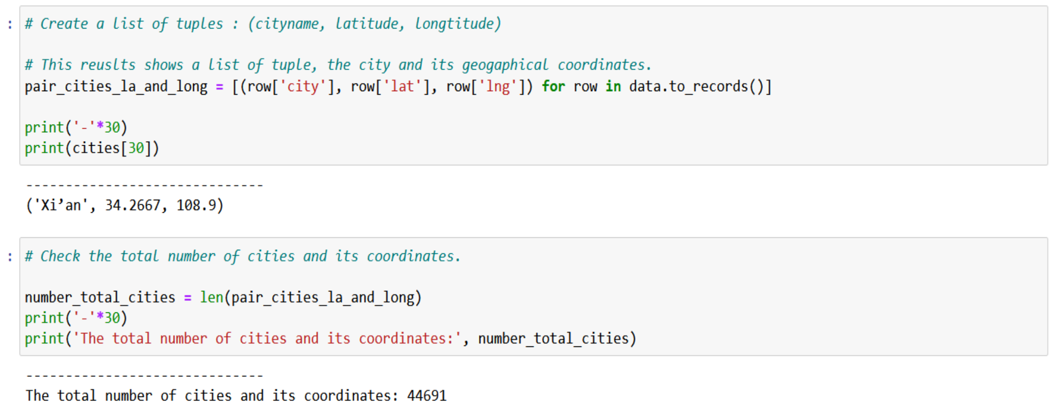
```
137
```

```
# Duplicate in the same city name
la_and_long = data['city']
la_and_long.duplicated().sum()
```

```
3348
```

Duplication Verification for Identical City Names and Geographical Coordinates:

1. Create a list of tuples



```
# Create a list of tuples : (cityname, latitude, longtitude)

# This reuslts shows a list of tuple, the city and its geogaphical coordinates.
pair_cities_la_and_long = [(row['city'], row['lat'], row['lng']) for row in data.to_records()]

print('-'*30)
print(cities[30])
```

```
------------------------------
('Xi’an', 34.2667, 108.9)
```

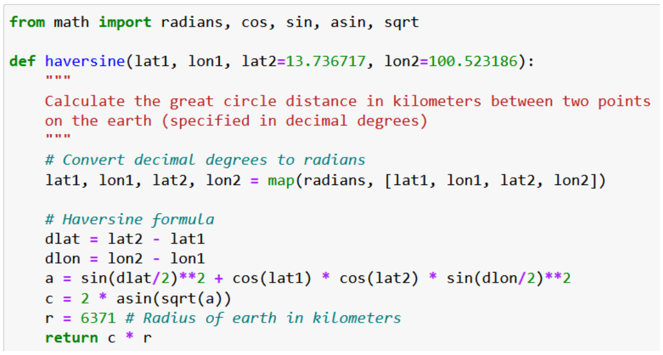
```
# Check the total number of cities and its coordinates.

number_total_cities = len(pair_cities_la_and_long)
print('-'*30)
print('The total number of cities and its coordinates:', number_total_cities)
```

```
------------------------------
The total number of cities and its coordinates: 44691
```

In this step, the following coordinate pairs were converted into tuples using a Python list comprehension method. The code generates a list named ‘pair_cities_l a_and_long,’ where each element is a tuple comprising the city name, latitude, and longitude.

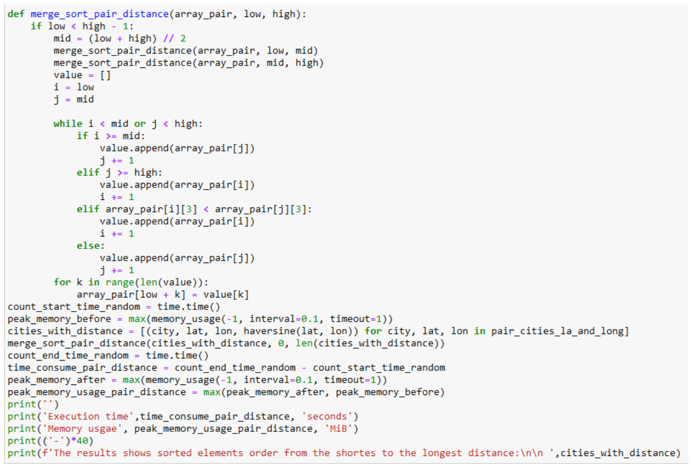
II. Sorting process

1) Merge sort Coordinates by Distance using Haversine Formula



```python
from math import radians, cos, sin, asin, sqrt

def haversine(lat1, lon1, lat2=13.736717, lon2=100.523186):
    """
    Calculate the great circle distance in kilometers between two points
    on the earth (specified in decimal degrees)
    """
    # Convert decimal degrees to radians
    lat1, lon1, lat2, lon2 = map(radians, [lat1, lon1, lat2, lon2])

    # Haversine formula
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    r = 6371 # Radius of earth in kilometers
    return c * r
```

The dataset has been arranged in ascending order based on the distance of each coordinate pair. From the original area (1) to the city of Bangkok (destination 2 – latitude=13.736717, longitude=100.523186), with the shortest distances showing first. To measure the distance between latitude and longitude coordinates, we utilized the Haversine formula, which is a mathematical theory that calculates the distance between two points. In the images above, the distance is showcased in kilometers. **The code implemented from (Stack Overflow. (n.d.). Python, 2023).**

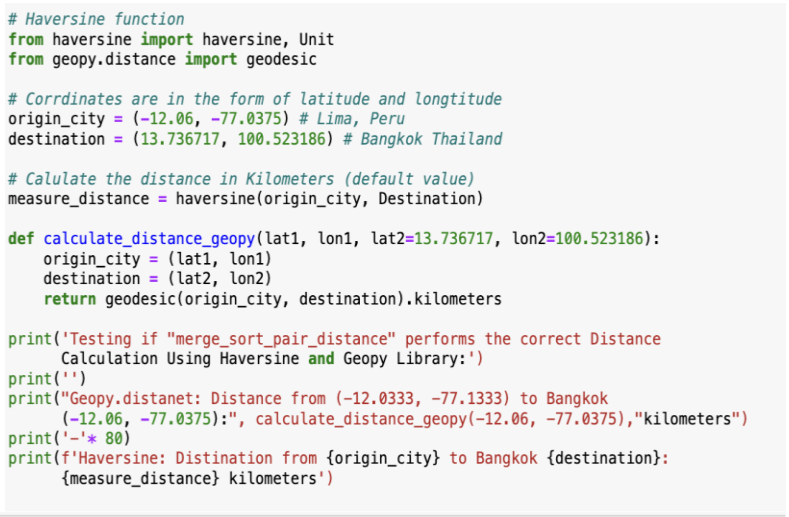


```python
def merge_sort_pair_distance(array_pair, low, high):
    if low < high - 1:
        mid = (low + high) // 2
        merge_sort_pair_distance(array_pair, low, mid)
        merge_sort_pair_distance(array_pair, mid, high)
        value = []
        i = low
        j = mid

        while i < mid or j < high:
            if i >= mid:
                value.append(array_pair[j])
                j += 1
            elif j >= high:
                value.append(array_pair[i])
                i += 1
            elif array_pair[i][3] < array_pair[j][3]:
                value.append(array_pair[i])
                i += 1
            else:
                value.append(array_pair[j])
                j += 1
        for k in range(len(value)):
            array_pair[low + k] = value[k]
count_start_time_random = time.time()
peak_memory_before = max(memory_usage(-1, interval=0.1, timeout=1))
cities_with_distance = [(city, lat, lon, haversine(lat, lon)) for city, lat, lon in pair_cities_la_and_long]
merge_sort_pair_distance(cities_with_distance, 0, len(cities_with_distance))
count_end_time_random = time.time()
time_consume_pair_distance = count_end_time_random - count_start_time_random
peak_memory_after = max(memory_usage(-1, interval=0.1, timeout=1))
peak_memory_usage_pair_distance = max(peak_memory_after, peak_memory_before)
print('')
print('Execution time',time_consume_pair_distance, 'seconds')
print('Memory usgae', peak_memory_usage_pair_distance, 'MiB')
print(('-')*40)
print(f'The results shows sorted elements order from the shortes to the longest distance:\n\n ',cities_with_distance)
```

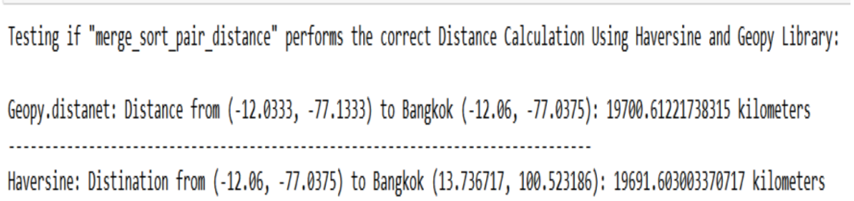
Testing and debugging

1) Evaluating the Accuracy of 'merge_sort_pair_distance()'

This method is used to confirm the accurateness of the Haversine function in calculating the distance between two geographical points. The evaluation involves two methods: the Haversine library and the geodesic function from the geopy library. Thereafter, we compare and assess their outputs. Lima, Peru, serves as an example in this evaluation and the result demonstrates the accuracy of the calculated distance from the 'merge_sort_pair_distance()' function.



```python
# Haversine function
from haversine import haversine, Unit
from geopy.distance import geodesic

# Corrdinates are in the form of latitude and longtitude
origin_city = (-12.06, -77.0375) # Lima, Peru
destination = (13.736717, 100.523186) # Bangkok Thailand

# Calulate the distance in Kilometers (default value)
measure_distance = haversine(origin_city, Destination)

def calculate_distance_geopy(lat1, lon1, lat2=13.736717, lon2=100.523186):
    origin_city = (lat1, lon1)
    destination = (lat2, lon2)
    return geodesic(origin_city, destination).kilometers

print('Testing if "merge_sort_pair_distance" performs the correct Distance
      Calculation Using Haversine and Geopy Library:')
print('')
print("Geopy.distanet: Distance from (-12.0333, -77.1333) to Bangkok
      (-12.06, -77.0375):", calculate_distance_geopy(-12.06, -77.0375),"kilometers")
print('-'* 80)
print(f'Haversine: Distination from {origin_city} to Bangkok {destination}:
      {measure_distance} kilometers')
```



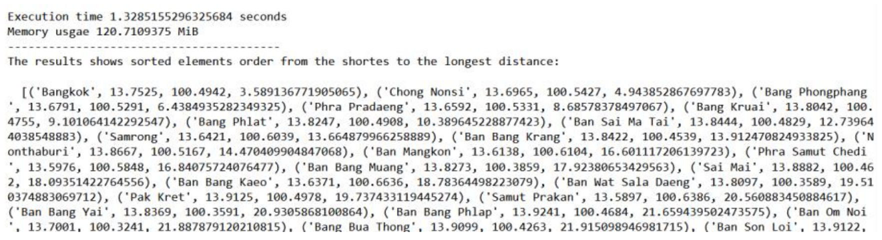
```
Testing if "merge_sort_pair_distance" performs the correct Distance Calculation Using Haversine and Geopy Library:

Geopy.distanet: Distance from (-12.0333, -77.1333) to Bangkok (-12.06, -77.0375): 19700.61221738315 kilometers
--------------------------------------------------------------------------------
Haversine: Distination from (-12.06, -77.0375) to Bangkok (13.736717, 100.523186): 19691.603003370717 kilometers
```

References:

- [Stack Overflow. (n.d.). python - Getting distance between two points based on latitude/longitude. [online] Available at: https://stackoverflow.com/questions/19412462/getting-distance-between-two-points-based-on-latitude-longitude.]
- [Louwers, J. (2023). Calculate Geographic distances in Python with the Haversine method. [online] Medium. Available at: https://louwersj.medium.com/calculate-geographic-distances-in-python-with-the-haversine-method-ed99b41ff04b.]

1-a) Observations

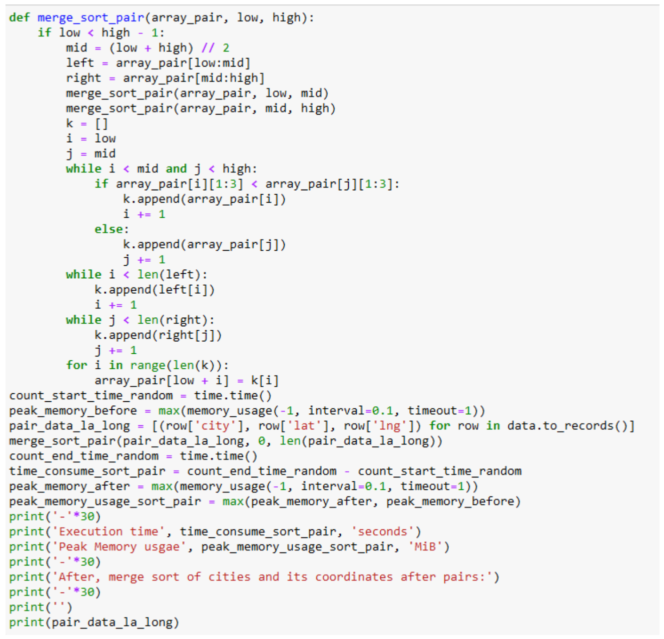
The merge sort algorithm computes distances of latitude and longitude coordinates to 'Bangkok,' while recording the time execution and memory usage for efficiency evaluation. While time consumption showed a small increase in time consumption and higher memory usage, could mean potential impacts from dataset size or computational complexity. We could further explore this and find out how to reduce memory consumption while maintaining speed.



```
Execution time 1.3285155296325684 seconds
Memory usgae 120.7109375 MiB
----------------------------------------
The results shows sorted elements order from the shortes to the longest distance:

  [('Bangkok', 13.7525, 100.4942, 3.589136771905065), ('Chong Nonsi', 13.6965, 100.5427, 4.943852867697783), ('Bang Phongphang
', 13.6791, 100.5291, 6.4384935282349325), ('Phra Pradaeng', 13.6592, 100.5331, 8.68578378497067), ('Bang Kruai', 13.8042, 100.
4755, 9.101064142292547), ('Bang Phlat', 13.8247, 100.4908, 10.389645228877423), ('Ban Sai Ma Tai', 13.8444, 100.4829, 12.73964
4038548883), ('Samrong', 13.6421, 100.6039, 13.664879966258889), ('Ban Bang Krang', 13.8422, 100.4539, 13.912470824933825), ('N
onthaburi', 13.8667, 100.5167, 14.470409904847068), ('Ban Mangkon', 13.6138, 100.6104, 16.601117206139723), ('Phra Samut Chedi
', 13.5976, 100.5848, 16.84075724076477), ('Ban Bang Muang', 13.8273, 100.3859, 17.92380653429563), ('Sai Mai', 13.8882, 100.46
2, 18.09351422764556), ('Ban Bang Kaeo', 13.6371, 100.6636, 18.78364498223079), ('Ban Wat Sala Daeng', 13.8097, 100.3589, 19.51
0374883069712), ('Pak Kret', 13.9125, 100.4978, 19.737433119445274), ('Samut Prakan', 13.5897, 100.6386, 20.560883450884617),
('Ban Bang Yai', 13.8369, 100.3591, 20.9305868100864), ('Ban Bang Phlap', 13.9241, 100.4684, 21.659439502473575), ('Ban Om Noi
', 13.7001, 100.3241, 21.887879120210815), ('Bang Bua Thong', 13.9099, 100.4263, 21.915098946981715), ('Ban Son Loi', 13.9122,
```

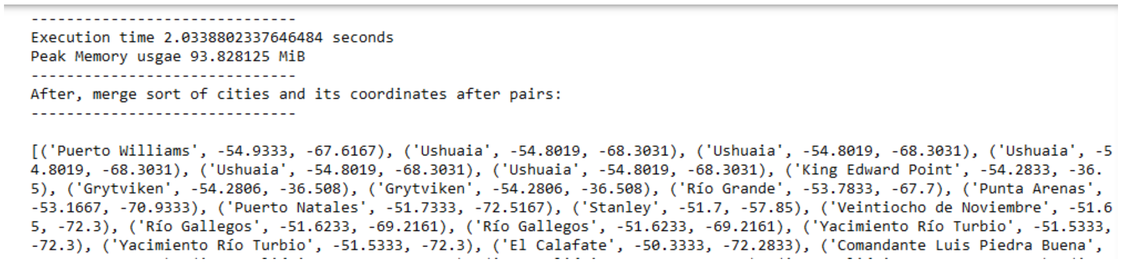
2) Additional exploration – Second approach: Sequential Sorting by latitude followed by longitude

As an exploratory step beyond the task requirements, we choose a secondary merge sorting approach that doesn't include distance measure calculation. This method sequentially sorts data by latitude and then longitude, reflecting the natural order of the cities' locations. The sorting algorithm ensures that the cities with the same latitude are arranged in their actual geographical order considering both latitude and longitude for two pairs.



```python
def merge_sort_pair(array_pair, low, high):
    if low < high - 1:
        mid = (low + high) // 2
        left = array_pair[low:mid]
        right = array_pair[mid:high]
        merge_sort_pair(array_pair, low, mid)
        merge_sort_pair(array_pair, mid, high)
        k = []
        i = low
        j = mid
        while i < mid and j < high:
            if array_pair[i][1:3] < array_pair[j][1:3]:
                k.append(array_pair[i])
                i += 1
            else:
                k.append(array_pair[j])
                j += 1
        while i < len(left):
            k.append(left[i])
            i += 1
        while j < len(right):
            k.append(right[j])
            j += 1
        for i in range(len(k)):
            array_pair[low + i] = k[i]
count_start_time_random = time.time()
peak_memory_before = max(memory_usage(-1, interval=0.1, timeout=1))
pair_data_la_long = [(row['city'], row['lat'], row['lng']) for row in data.to_records()]
merge_sort_pair(pair_data_la_long, 0, len(pair_data_la_long))
count_end_time_random = time.time()
time_consume_sort_pair = count_end_time_random - count_start_time_random
peak_memory_after = max(memory_usage(-1, interval=0.1, timeout=1))
peak_memory_usage_sort_pair = max(peak_memory_after, peak_memory_before)
print('-'*30)
print('Execution time', time_consume_sort_pair, 'seconds')
print('Peak Memory usgae', peak_memory_usage_sort_pair, 'MiB')
print('-'*30)
print('After, merge sort of cities and its coordinates after pairs:')
print('-'*30)
print('')
print(pair_data_la_long)
```

Observations: Sorting sequentially by latitude followed by longitude showcases better efficiency in memory usage, lesser in peak memory usage than (1-a) while time execution shows small increases, suggesting that because this algorithm sorts data without additional computational processes.

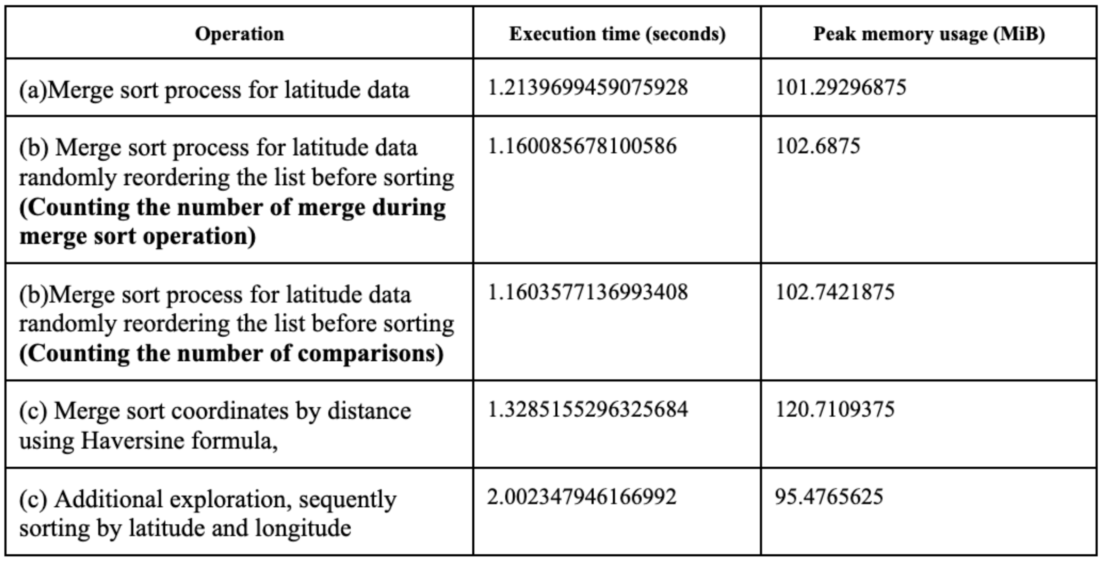


```
------------------------------
Execution time 2.0338802337646484 seconds
Peak Memory usgae 93.828125 MiB
------------------------------
After, merge sort of cities and its coordinates after pairs:
------------------------------

[('Puerto Williams', -54.9333, -67.6167), ('Ushuaia', -54.8019, -68.3031), ('Ushuaia', -54.8019, -68.3031), ('Ushuaia', -5
4.8019, -68.3031), ('Ushuaia', -54.8019, -68.3031), ('Ushuaia', -54.8019, -68.3031), ('King Edward Point', -54.2833, -36.
5), ('Grytviken', -54.2806, -36.508), ('Grytviken', -54.2806, -36.508), ('Río Grande', -53.7833, -67.7), ('Punta Arenas',
-53.1667, -70.9333), ('Puerto Natales', -51.7333, -72.5167), ('Stanley', -51.7, -57.85), ('Veintiocho de Noviembre', -51.6
5, -72.3), ('Río Gallegos', -51.6233, -69.2161), ('Río Gallegos', -51.6233, -69.2161), ('Yacimiento Río Turbio', -51.5333,
-72.3), ('Yacimiento Río Turbio', -51.5333, -72.3), ('El Calafate', -50.3333, -72.2833), ('Comandante Luis Piedra Buena',
```

Overall Observation:

1) **Compare time and memory execution in each merge sort process:**

- **Consistent in execution and memory usage for basic Merge sort:** Both the basic merge sort (a) and the added complexity of counting merges in randomly ordered latitude datasets (b) showcase similar and consistent execution times, with identical memory usage. This suggests that the additional step of reordering the latitude dataset does not significantly increase the algorithm's memory requirements. Similarly, the execution time does not show a substantial rise, meaning that the random reordering process does not significantly impede the sorting efficiency.



Operation	Execution time (seconds)	Peak memory usage (MiB)
(a)Merge sort process for latitude data	1.2139699459075928	101.29296875
(b) Merge sort process for latitude data randomly reordering the list before sorting **(Counting the number of merge during merge sort operation)**	1.160085678100586	102.6875
(b)Merge sort process for latitude data randomly reordering the list before sorting **(Counting the number of comparisons)**	1.1603577136993408	102.7421875
(c) Merge sort coordinates by distance using Haversine formula,	1.3285155296325684	120.7109375
(c) Additional exploration, sequently sorting by latitude and longitude	2.002347946166992	95.4765625

- **Increased memory usage with complexity**: Memory usage showcases a slight increase when sorting coordinates based on distance using the Haversine Formula calculation. This increase in memory usage may be attributed to the inclusion of additional data structures (latitude and longitude) or to the computational complexity involved in distance calculations. This raises questions about scalability and particularly when handling large datasets.

- **Well performance in Sequential Sorting:** Sorting sequentially by latitude and then by longitude shows that the execution time and memory usage are compatible with the basic merge sort operation in both (a) and (b) even with longer execution time. It's important to note that this method does not affect the performance negatively in terms of time.

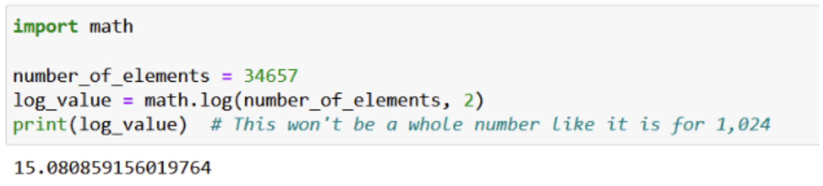
2) Complexity analysis of merge sort.

Determining the exact time and space requirements for running this function was challenging due to various factors in the machine environment and concurrent processes. Therefore, instead of the original question we asked, "How does the runtime of this function grow with an increase in size?" Time complexity is used to solve this problem, because it expresses how the algorithm's runtime scales with the output size by using the mathematical notation such as Big O notation.

Example from (a):

- In O(n*log n), the big O notation sets an upper limit on the algorithm's runtime. In basic merge sort (a), where n is the input size (34,657 in this case), log n represents how many times the input can be divided by 2 until reaching down to one. In this code, the logarithm is calculated during the merge sort, with approximately 15 divisions for n = 34,657 (log2(34657) ≈ 15.08). Meaning that in the input size of 34,657, the data can be divided into subarrays approximately 15 times before reaching the small unit. The same calculation as the input size of latitude and longitude data n = 44,691 (log2(44691) ≈ 15.447).

Giving an example of basic merge sort (a), where the base 2 of the algorithm is 34,657 which is approximately 15.08 and (b) the base 2 of the algorithm is 44,691, approximately 15.44 times. This shows that, even with a substantial increase in data size, the number of levels of division (the depth of the recursion tree) doesn't grow rapidly. This is because of the O(n log n) nature of the algorithm, where the number of necessary comparisons keeps the overall complexity quite efficient. The growth rate remains proportional to n*log(n), which is notably slower than exponential growth.



```
import math

number_of_elements = 34657
log_value = math.log(number_of_elements, 2)
print(log_value)  # This won't be a whole number like it is for 1,024
```

```
15.080859156019764
```

Given our understanding of complexity in basic merge sort (a), we'll break it down into each and analyze the differences:

Merge sort implementation of latitude data:

- Time complexity (O(n log n)): With 34,657 latitude values, the time complexity aligns with the known O(n log n) complexity.
- Space complexity (O(n log n)): Space varies between (a) and (b), with (c) indicating increased space usage due to the merging process. The space complexity is O(n), corresponding to the array size.
- Counting the number of merges: In (b), the number of merge operations aligns with the height of the merge sort tree which is a fixed property regardless of data.

Merge sort with Haversine Formula:

- Time complexity (O(n log n)): Calculating distances using the Haversine formula adds complexity in (C). However, as these calculations are done once per element pair and stored, the overall complexity remains O(n log n).
- Space Complexity: In (C), space usage rises, particularly if distances are sorted in an additional array and computation in each sub pair array, which can potentially double the space and add another O(n) during the process. In that case, Merge sort requires additional memory to store the merged sub-arrays during the sorting process (GeeksforGeeks, 2023).

Secondary, sequential sorting by latitude and longitude:

- Time complexity (O(n log n)): Sorting twice under certain conditions in (C) doesn't fundamentally change the sorting method, impacting execution time and space usage comparably to other merge sort functions.
- Space complexity (O(n)): Similar to the basic merge sort (a), the space complexity remains O(n).

2) Quick Sort Problem

The Quick Sort algorithm is a sorting algorithm employs the divide-and-conquer concept:

Pivot Selection: First it chooses a pivot element. Thereafter, it partitions the data into two subarrays, with elements in the left subarray are less than or equal to the pivot and elements in the right subarray are greater than the pivot, which is positioned in the middle.
Recursion: Recursively sorts the two subarrays.
Combination: Combines the sorted arrays into a single array.

Pivot selection is a crucial step therefore in this report, we'll explore the commonly used strategy of random pivot selection. Additionally, for comparative analysis, the method of selecting the pivot from the last index of the array will also be implemented.

Decision-Making: To understand the influence of pivot selection, we employed two approaches:

Random Pivot: The pivot value is chosen based on a random number.
Last Element as Pivot: The pivot is selected from the last index of the array.

The choice of pivot value significantly impacts the sorting process, providing a generally faster implementation of Quicksort and reducing the probability of worst-case scenarios. Opting for a random pivot reduces the chance of encountering worst-case O(n^2) performance. While selecting the first element as the pivot may be simple but it often results in poor performance, as all elements end up in one partition. Hence, for comparative analysis, we decided to sort the data using both random and last element as a pivot approach.
Our approach to the quicksort algorithm in this report follows these steps:

Initiate Quick Sort: Starts with an unsorted array, indexed from 0 to n-1, where n is the number of elements.

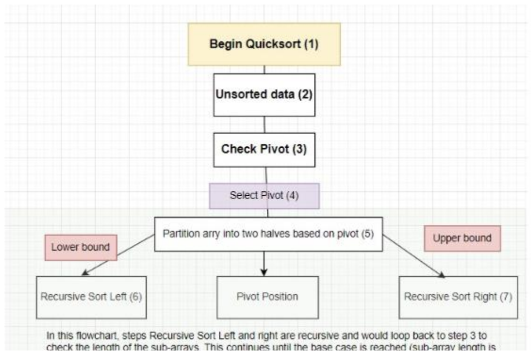
Pivot Selection: Randomly selects a pivot element, often chosen from the rightmost element of the array.

Partitioning: Here the array with a start index moves from left to right and the end index moves from right to left. If the element at the start index is greater than the pivot, the start position stops increasing.

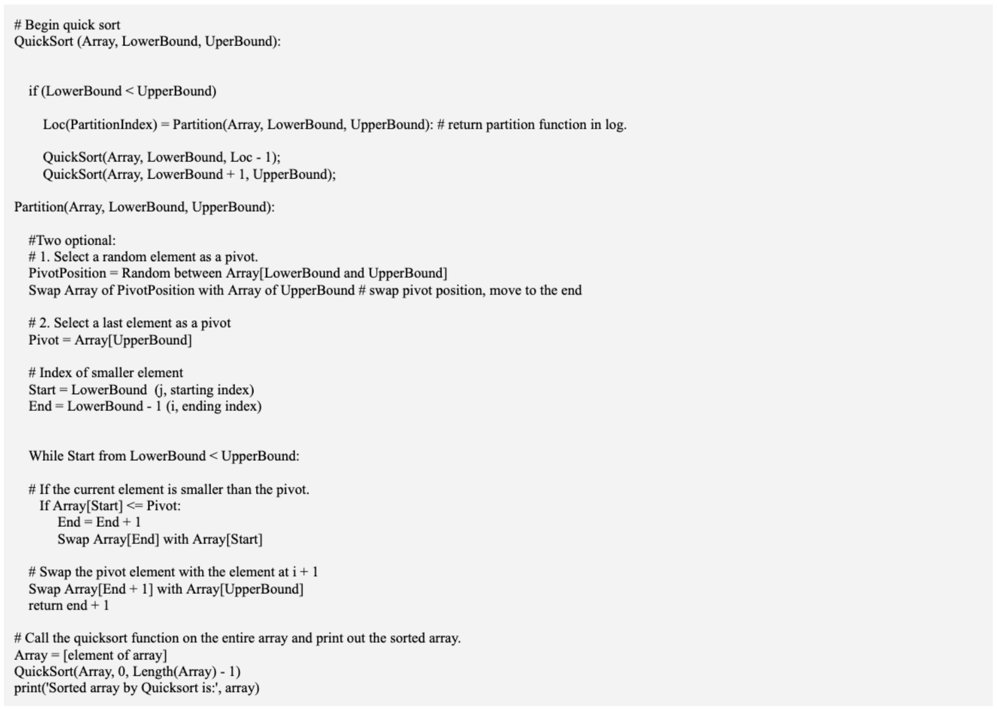
Similarly, if an element at the end index is less than or equal to the pivot, the end position stops decreasing. Elements will then be swapped until conditions are met. When both start and end positions have stopped, if the start position is less than the end position, elements are swapped to ensure the left side is smaller than the pivot, with larger indexes on the right.

Recursion: The algorithm recursively sorts the subarrays formed by the partitioning step.

The flowchart below visualizes how quicksort works during the sorting process :


Begin Quicksort (1)
Unsorted data (2)
Check Pivot (3)
Select Pivot (4)
Partition arry into two halves based on pivot (5)
Lower bound
Upper bound
Recursive Sort Left (6)
Pivot Position
Recursive Sort Right (7)
In this flowchart, steps Recursive Sort Left and right are recursive and would loop back to step 3 to
check the length of the sub-arrays. This continues until the base case is reached (sub-array length is

II. Pseudo code



```
# Begin quick sort
QuickSort (Array, LowerBound, UperBound):


    if (LowerBound < UpperBound)

        Loc(PartitionIndex) = Partition(Array, LowerBound, UpperBound): # return partition function in log.

        QuickSort(Array, LowerBound, Loc - 1);
        QuickSort(Array, LowerBound + 1, UpperBound);

Partition(Array, LowerBound, UpperBound):

    #Two optional:
    # 1. Select a random element as a pivot.
    PivotPosition = Random between Array[LowerBound and UpperBound]
    Swap Array of PivotPosition with Array of UpperBound # swap pivot position, move to the end

    # 2. Select a last element as a pivot
    Pivot = Array[UpperBound]

    # Index of smaller element
    Start = LowerBound  (j, starting index)
    End = LowerBound - 1 (i, ending index)


    While Start from LowerBound < UpperBound:

    # If the current element is smaller than the pivot.
       If Array[Start] <= Pivot:
           End = End + 1
           Swap Array[End] with Array[Start]

    # Swap the pivot element with the element at i + 1
    Swap Array[End + 1] with Array[UpperBound]
    return end + 1

# Call the quicksort function on the entire array and print out the sorted array.
Array = [element of array]
QuickSort(Array, 0, Length(Array) - 1)
print('Sorted array by Quicksort is:', array)
```

Positioning the Pivot: Finally, it places the pivot in the correct sorting position. The recursion loop stops when the subarrays are fully reduced to single elements which means the entire array is sorted.
The code shown in the image showcases how the quicksort algorithm is implemented in Python.The Partition function determines the correct pivot location, while the QuickSort function recursively applies the algorithm to subarrays.The pivot in the Partition function is either a randomly selected element or the last element. The array is then partitioned, and the pivot is correctly placed in the sorted array. The sorted array is the outcome of QuickSort, that gets implemented after an array has been initialized in the main function. QuickSort is an effective sorting algorithm with an average time complexity of O(n log n).

a) **Implement a proper quick sort algorithm so that all city latitudes are in an ordered list.**

To solve the question (a), we chose to employ two strategies in implementing the Quicksort Algorithm: Random Pivot Selection and Fixed Pivot Position from the Last Index.

I. Random Pivot Selection:

Rationale: Selecting a pivot randomly is recommended by various sources. It enhances the sorting process, diminishing the likelihood of worst-case scenarios. This method is particularly beneficial when dealing with datasets that may already have some values sorted in place.

II. Fixed Pivot Position (Last Index):

Rationale: Selecting the last element of the index as a pivot is a common approach. It is widely used and very efficient in practical applications. However, it may exhibit slower performance when sorting large datasets.

I. Random Pivot Selection:

The code in the image showcases Quicksort with a random pivot selection. It measures the algorithm's performance in terms of time, memory usage, and comparisons. The result will be the sorted array.

Here's a breakdown:

quicksort_randomPivot Function:

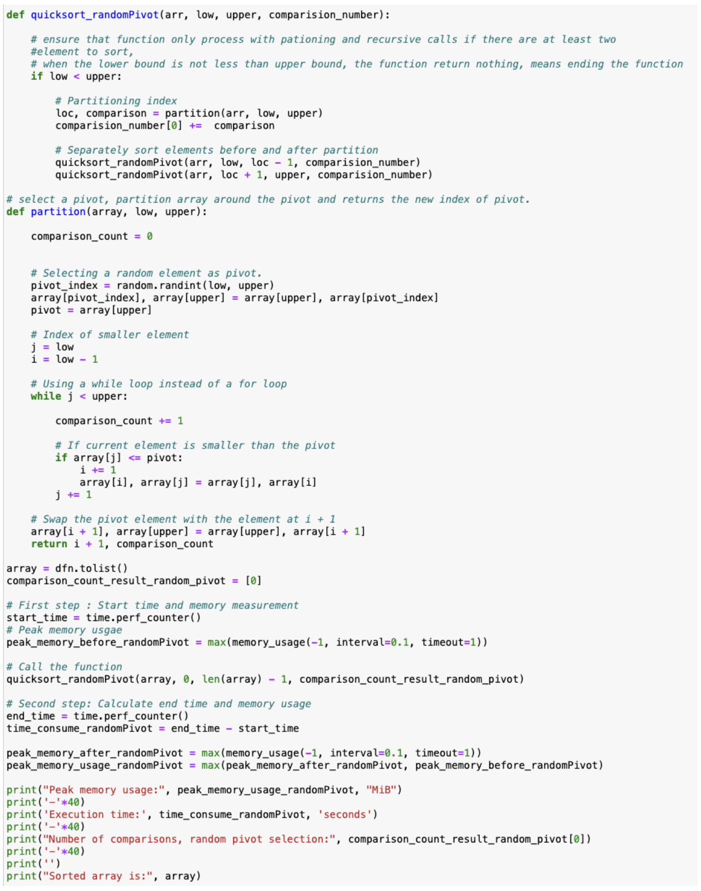
- Sorts the array with a random pivot.
- Recursively applies the algorithm to subarrays.
- Counts the number of comparisons made during sorting.

partition Function:

- Selects a random pivot and places it correctly in the array.
- Counts the number of comparisons during partitioning.

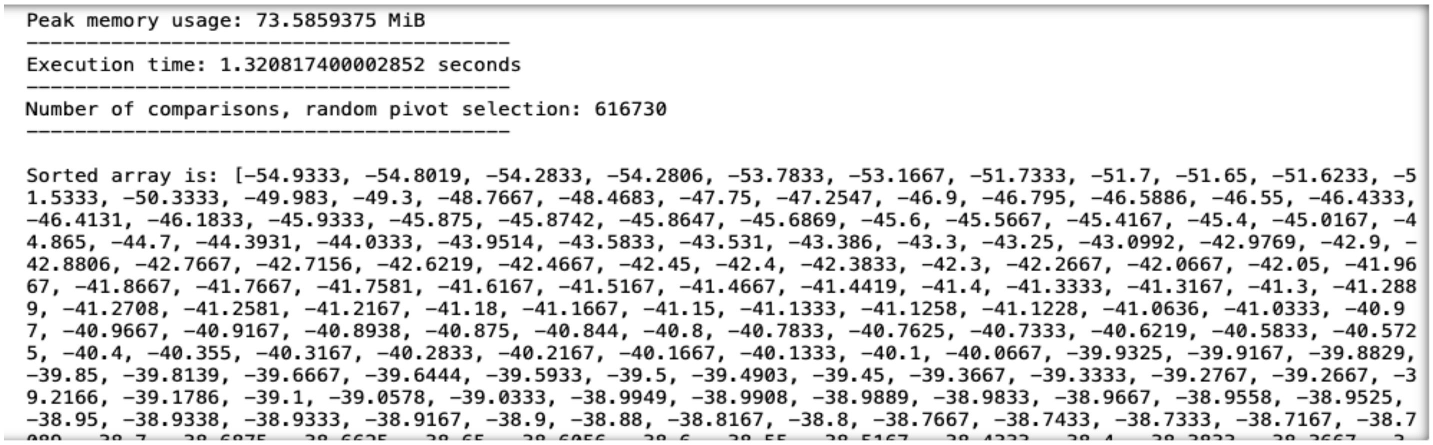
Main Code:

- Converts the input array to a Python list.
- Measures peak memory usage and records the start time.
- Calls quicksort_randomPivot.
- Records end time and peak memory usage after sorting.
- Prints peak memory, execution time, comparisons, and the sorted array.



```python
def quicksort_randomPivot(arr, low, upper, comparision_number):

    # ensure that function only process with pationing and recursive calls if there are at least two
    #element to sort,
    # when the lower bound is not less than upper bound, the function return nothing, means ending the function
    if low < upper:

        # Partitioning index
        loc, comparison = partition(arr, low, upper)
        comparision_number[0] +=  comparison

        # Separately sort elements before and after partition
        quicksort_randomPivot(arr, low, loc - 1, comparision_number)
        quicksort_randomPivot(arr, loc + 1, upper, comparision_number)

# select a pivot, partition array around the pivot and returns the new index of pivot.
def partition(array, low, upper):

    comparison_count = 0


    # Selecting a random element as pivot.
    pivot_index = random.randint(low, upper)
    array[pivot_index], array[upper] = array[upper], array[pivot_index]
    pivot = array[upper]

    # Index of smaller element
    j = low
    i = low - 1

    # Using a while loop instead of a for loop
    while j < upper:

        comparison_count += 1

        # If current element is smaller than the pivot
        if array[j] <= pivot:
            i += 1
            array[i], array[j] = array[j], array[i]
        j += 1

    # Swap the pivot element with the element at i + 1
    array[i + 1], array[upper] = array[upper], array[i + 1]
    return i + 1, comparison_count

array = dfn.tolist()
comparison_count_result_random_pivot = [0]

# First step : Start time and memory measurement
start_time = time.perf_counter()
# Peak memory usgae
peak_memory_before_randomPivot = max(memory_usage(-1, interval=0.1, timeout=1))

# Call the function
quicksort_randomPivot(array, 0, len(array) - 1, comparison_count_result_random_pivot)

# Second step: Calculate end time and memory usage
end_time = time.perf_counter()
time_consume_randomPivot = end_time - start_time

peak_memory_after_randomPivot = max(memory_usage(-1, interval=0.1, timeout=1))
peak_memory_usage_randomPivot = max(peak_memory_after_randomPivot, peak_memory_before_randomPivot)

print("Peak memory usage:", peak_memory_usage_randomPivot, "MiB")
print('-'*40)
print('Execution time:', time_consume_randomPivot, 'seconds')
print('-'*40)
print("Number of comparisons, random pivot selection:", comparison_count_result_random_pivot[0])
print('-'*40)
print('')
print("Sorted array is:", array)
```

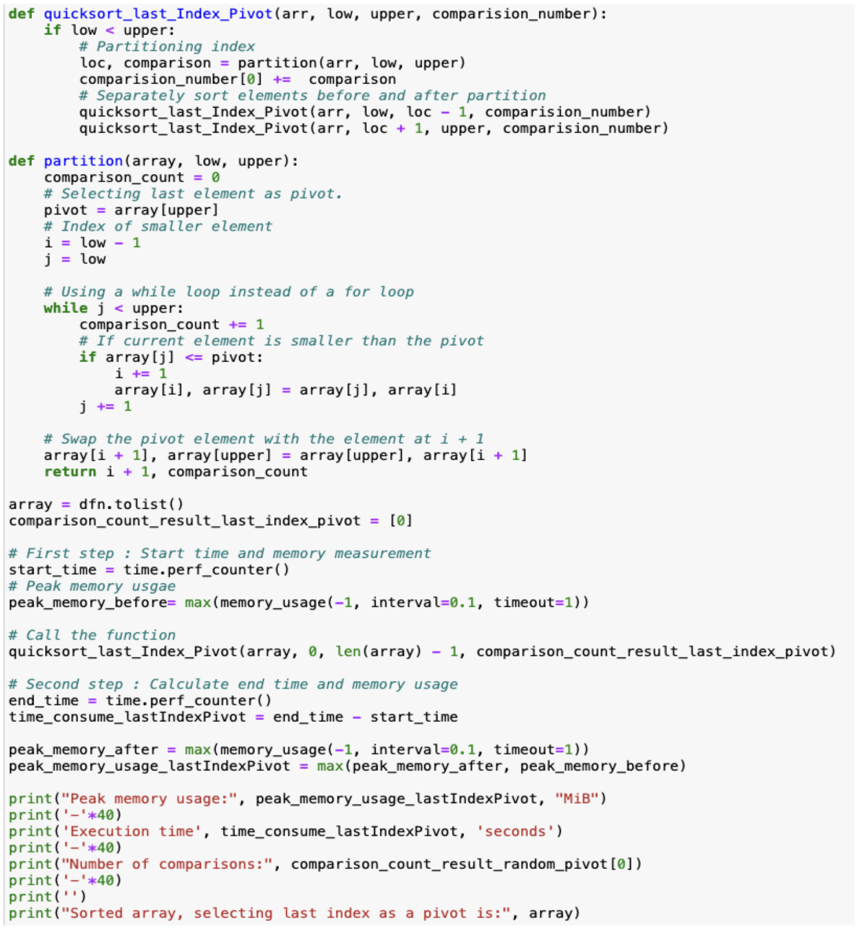
The result is an efficient sorting algorithm with insights into its performance metrics.



```
Peak memory usage: 73.5859375 MiB
----------------------------------------
Execution time: 1.320817400002852 seconds
----------------------------------------
Number of comparisons, random pivot selection: 616730
----------------------------------------

Sorted array is: [-54.9333, -54.8019, -54.2833, -54.2806, -53.7833, -53.1667, -51.7333, -51.7, -51.65, -51.6233, -5
1.5333, -50.3333, -49.983, -49.3, -48.7667, -48.4683, -47.75, -47.2547, -46.9, -46.795, -46.5886, -46.55, -46.4333,
-46.4131, -46.1833, -45.9333, -45.875, -45.8742, -45.8647, -45.6869, -45.6, -45.5667, -45.4167, -45.4, -45.0167, -4
4.865, -44.7, -44.3931, -44.0333, -43.9514, -43.5833, -43.531, -43.386, -43.3, -43.25, -43.0992, -42.9769, -42.9, -
42.8806, -42.7667, -42.7156, -42.6219, -42.4667, -42.45, -42.4, -42.3833, -42.3, -42.2667, -42.0667, -42.05, -41.96
67, -41.8667, -41.7667, -41.7581, -41.6167, -41.5167, -41.4667, -41.4419, -41.4, -41.3333, -41.3167, -41.3, -41.288
9, -41.2708, -41.2581, -41.2167, -41.18, -41.1667, -41.15, -41.1333, -41.1258, -41.1228, -41.0636, -41.0333, -40.9
7, -40.9667, -40.9167, -40.8938, -40.875, -40.844, -40.8, -40.7833, -40.7625, -40.7333, -40.6219, -40.5833, -40.572
5, -40.4, -40.355, -40.3167, -40.2833, -40.2167, -40.1667, -40.1333, -40.1, -40.0667, -39.9325, -39.9167, -39.8829,
-39.85, -39.8139, -39.6667, -39.6444, -39.5933, -39.5, -39.4903, -39.45, -39.3667, -39.3333, -39.2767, -39.2667, -3
9.2166, -39.1786, -39.1, -39.0578, -39.0333, -38.9949, -38.9908, -38.9889, -38.9833, -38.9667, -38.9558, -38.9525,
-38.95, -38.9338, -38.9333, -38.9167, -38.9, -38.88, -38.8167, -38.8, -38.7667, -38.7433, -38.7333, -38.7167, -38.7
```

II. Approach: Fixed Pivot Position (Last Index):

The code shown in the images implements Quicksort with a fixed pivot strategy, choosing the last element as the pivot. Here's a breakdown:



```python
def quicksort_last_Index_Pivot(arr, low, upper, comparision_number):
    if low < upper:
        # Partitioning index
        loc, comparison = partition(arr, low, upper)
        comparision_number[0] +=  comparison
        # Separately sort elements before and after partition
        quicksort_last_Index_Pivot(arr, low, loc - 1, comparision_number)
        quicksort_last_Index_Pivot(arr, loc + 1, upper, comparision_number)

def partition(array, low, upper):
    comparison_count = 0
    # Selecting last element as pivot.
    pivot = array[upper]
    # Index of smaller element
    i = low - 1
    j = low

    # Using a while loop instead of a for loop
    while j < upper:
        comparison_count += 1
        # If current element is smaller than the pivot
        if array[j] <= pivot:
            i += 1
            array[i], array[j] = array[j], array[i]
        j += 1

    # Swap the pivot element with the element at i + 1
    array[i + 1], array[upper] = array[upper], array[i + 1]
    return i + 1, comparison_count

array = dfn.tolist()
comparison_count_result_last_index_pivot = [0]

# First step : Start time and memory measurement
start_time = time.perf_counter()
# Peak memory usgae
peak_memory_before= max(memory_usage(-1, interval=0.1, timeout=1))

# Call the function
quicksort_last_Index_Pivot(array, 0, len(array) - 1, comparison_count_result_last_index_pivot)

# Second step : Calculate end time and memory usage
end_time = time.perf_counter()
time_consume_lastIndexPivot = end_time - start_time

peak_memory_after = max(memory_usage(-1, interval=0.1, timeout=1))
peak_memory_usage_lastIndexPivot = max(peak_memory_after, peak_memory_before)

print("Peak memory usage:", peak_memory_usage_lastIndexPivot, "MiB")
print('-'*40)
print('Execution time', time_consume_lastIndexPivot, 'seconds')
print('-'*40)
print("Number of comparisons:", comparison_count_result_random_pivot[0])
print('-'*40)
print('')
print("Sorted array, selecting last index as a pivot is:", array)
```

<u>quicksort_last_Index_Pivot Function:</u>

- Sorts the array using the last element as the pivot.
- Recursively applies the algorithm to subarrays.
- Counts comparisons during sorting.

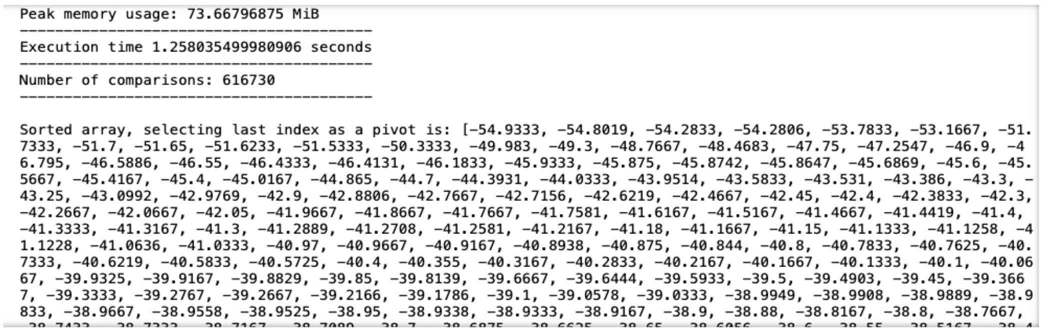
<u>partition Function:</u>

- Selects the last element as the pivot.
- Iterates through the array, swapping elements to arrange smaller ones on the left.
- Returns the partition index and the count of comparisons made.

Main Code:

- Converts the input array to a list.
- Measures peak memory usage and records start time.
- Calls quicksort_last_Index_Pivot.
- Records the end time and peak memory usage.
- Prints peak memory, execution time, comparisons, and the sorted array.

The result is a sorted array with detailed performance metrics for the algorithm.



```
Peak memory usage: 73.66796875 MiB
----------------------------------------
Execution time 1.258035499980906 seconds
----------------------------------------
Number of comparisons: 616730
----------------------------------------

Sorted array, selecting last index as a pivot is: [-54.9333, -54.8019, -54.2833, -54.2806, -53.7833, -53.1667, -51.
7333, -51.7, -51.65, -51.6233, -51.5333, -50.3333, -49.983, -49.3, -48.7667, -48.4683, -47.75, -47.2547, -46.9, -4
6.795, -46.5886, -46.55, -46.4333, -46.4131, -46.1833, -45.9333, -45.875, -45.8742, -45.8647, -45.6869, -45.6, -45.
5667, -45.4167, -45.4, -45.0167, -44.865, -44.7, -44.3931, -44.0333, -43.9514, -43.5833, -43.531, -43.386, -43.3, -
43.25, -43.0992, -42.9769, -42.9, -42.8806, -42.7667, -42.7156, -42.6219, -42.4667, -42.45, -42.4, -42.3833, -42.3,
-42.2667, -42.0667, -42.05, -41.9667, -41.8667, -41.7667, -41.7581, -41.6167, -41.5167, -41.4667, -41.4419, -41.4,
-41.3333, -41.3167, -41.3, -41.2889, -41.2708, -41.2581, -41.2167, -41.18, -41.1667, -41.15, -41.1333, -41.1258, -4
1.1228, -41.0636, -41.0333, -40.97, -40.9667, -40.9167, -40.8938, -40.875, -40.844, -40.8, -40.7833, -40.7625, -40.
7333, -40.6219, -40.5833, -40.5725, -40.4, -40.355, -40.3167, -40.2833, -40.2167, -40.1667, -40.1333, -40.1, -40.06
67, -39.9325, -39.9167, -39.8829, -39.85, -39.8139, -39.6667, -39.6444, -39.5933, -39.5, -39.4903, -39.45, -39.366
7, -39.3333, -39.2767, -39.2667, -39.2166, -39.1786, -39.1, -39.0578, -39.0333, -38.9949, -38.9908, -38.9889, -38.9
833, -38.9667, -38.9558, -38.9525, -38.95, -38.9338, -38.9333, -38.9167, -38.9, -38.88, -38.8167, -38.8, -38.7667,
```

Analysis: Method 1 vs Method 2

A) Quick Sort Process

The QuickSort algorithm was applied to a latitude array using two pivot selection methods: Random pivot and fixed pivot position (last index). Both methods produced identical results, confirming that the correctness of sorting is unaffected by the pivot selection method.

Divide (Step 1):

Both methods start by choosing an element in the subarray of latitude data as a pivot (utilizing both random number (a-1) and the last element (a-2) as examples). The algorithm rearranges elements, moving those less than the pivot to the left and those greater to the right, without sorting anything.

Recursive and Partition (Step 2):

This step recursively sorts subarrays until all elements on the left are less than or equal to the pivot, and those on the right are greater. Partitioning a subarray takes linear time (O(n)) as it depends only on individual elements, not the total number of elements.

Combine (Step 3):

After recursive sorts, all elements less than the pivot are on the right, and those greater than the pivot are on the left.

Performance Evaluation: Time Complexity & Memory Usage

- Pivot Decision:
    - **Random Pivot:** Reduces the worst-case scenario for QuickSort (O(n^2)), specially in scenarios with unbalanced partitions.
    - **Fixed Pivot (Last Index):** Can lead to worst-case complexity when the array is already sorted (O(n^2)).
- Execution Time:
    - Similar execution times for both strategies.

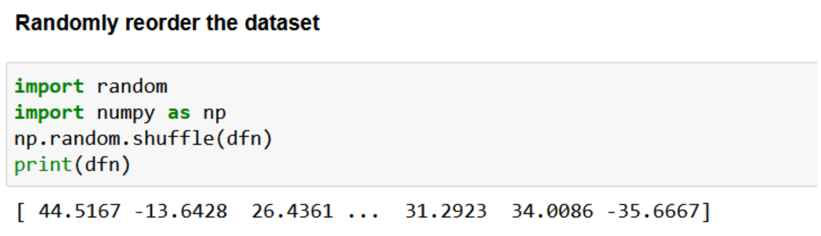
- Memory Usage:

    - Slightly higher memory usage with the random pivot compared to the last element pivot.
    - This difference may be due to random number generation and potential influences from concurrent processes.
- Number of Comparisons:
    - Slightly different number of comparisons, indicating no significant difference in overall performance.

In conclusion, both pivot selection methods correctly sort the latitude data with negligible variations in execution time and memory usage. The results shows the efficiency of QuickSort in the fast implementation and so in the minimal memory usage.

b) Count the number of comparisons needed to sort the dataset. Does it change if you randomly order the list before sorting? Why/why not?

Randomly reordering the dataset



Randomly reorder the dataset

```
import random
import numpy as np
np.random.shuffle(dfn)
print(dfn)
```

```
[ 44.5167 -13.6428  26.4361 ...  31.2923  34.0086 -35.6667]
```

Apply quicksort to see if the number of comparisons needed to change:

This code shown in the image utilizes the quicksort algorithm with a random pivot strategy for sorting a dataset.

Key points: Quicksort Function (quicksort_random_dataset):

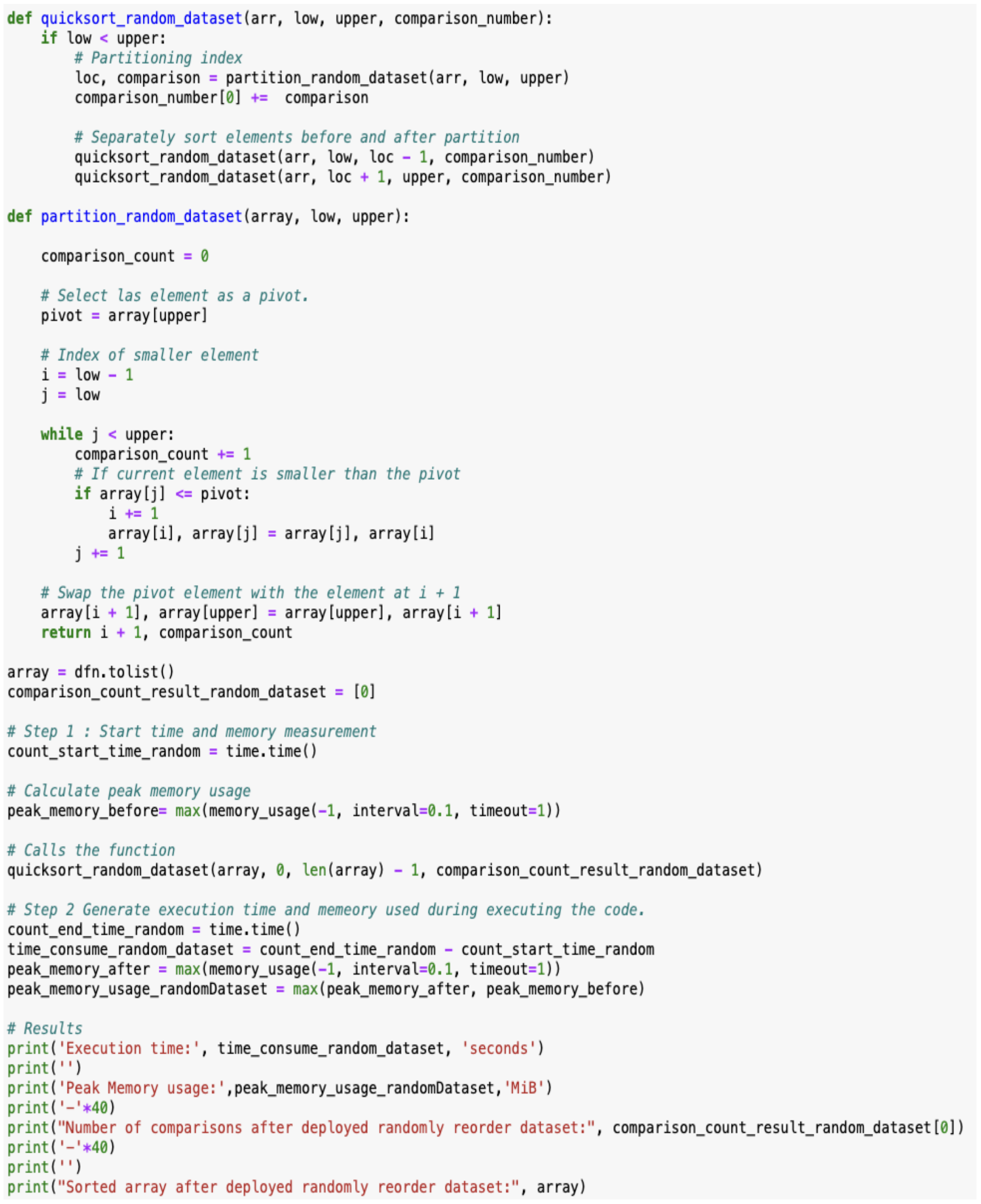
Recursively sorts subarrays using the quicksort algorithm and employs a randomly selected element as the pivot. Additionally, it tracks the number of comparisons during partitioning.

Partition Function (partition_random_dataset):

Partitions the array based on a randomly chosen pivot, thereafter it iIterates it through the array, ensuring a proper placement of elements relative to the pivot. At the end it returns the pivot's correct index and the number of comparisons.

Main Execution:

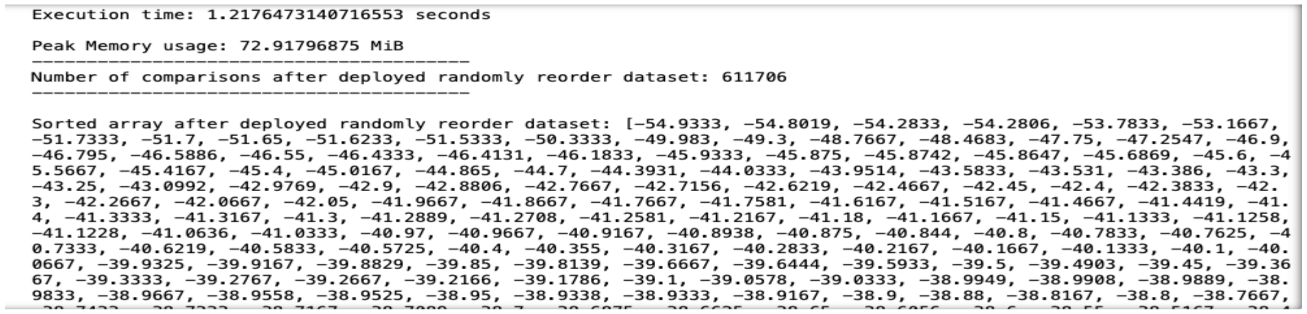
Initializes an array from a dataframe (dfn) and tracks the comparison count during quicksort for analysis.



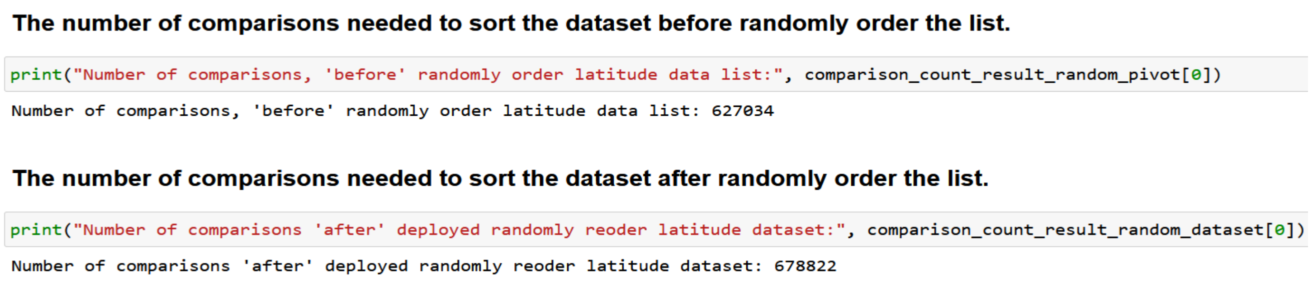
```
def quicksort_random_dataset(arr, low, upper, comparison_number):
    if low < upper:
        # Partitioning index
        loc, comparison = partition_random_dataset(arr, low, upper)
        comparison_number[0] +=  comparison

        # Separately sort elements before and after partition
        quicksort_random_dataset(arr, low, loc - 1, comparison_number)
        quicksort_random_dataset(arr, loc + 1, upper, comparison_number)

def partition_random_dataset(array, low, upper):

    comparison_count = 0

    # Select las element as a pivot.
    pivot = array[upper]

    # Index of smaller element
    i = low - 1
    j = low

    while j < upper:
        comparison_count += 1
        # If current element is smaller than the pivot
        if array[j] <= pivot:
            i += 1
            array[i], array[j] = array[j], array[i]
        j += 1

    # Swap the pivot element with the element at i + 1
    array[i + 1], array[upper] = array[upper], array[i + 1]
    return i + 1, comparison_count

array = dfn.tolist()
comparison_count_result_random_dataset = [0]

# Step 1 : Start time and memory measurement
count_start_time_random = time.time()

# Calculate peak memory usage
peak_memory_before= max(memory_usage(-1, interval=0.1, timeout=1))

# Calls the function
quicksort_random_dataset(array, 0, len(array) - 1, comparison_count_result_random_dataset)

# Step 2 Generate execution time and memeory used during executing the code.
count_end_time_random = time.time()
time_consume_random_dataset = count_end_time_random - count_start_time_random
peak_memory_after = max(memory_usage(-1, interval=0.1, timeout=1))
peak_memory_usage_randomDataset = max(peak_memory_after, peak_memory_before)

# Results
print('Execution time:', time_consume_random_dataset, 'seconds')
print('')
print('Peak Memory usage:',peak_memory_usage_randomDataset,'MiB')
print('-'*40)
print("Number of comparisons after deployed randomly reorder dataset:", comparison_count_result_random_dataset[0])
print('-'*40)
print('')
print("Sorted array after deployed randomly reorder dataset:", array)
```

Performance Measurement:

Measures execution time and peak memory usage before and after the quicksort, thereafter it prints results, including execution time, peak memory usage, comparisons count, and the sorted array.



```
Execution time: 1.2176473140716553 seconds

Peak Memory usage: 72.91796875 MiB
----------------------------------------
Number of comparisons after deployed randomly reorder dataset: 611706
----------------------------------------

Sorted array after deployed randomly reorder dataset: [-54.9333, -54.8019, -54.2833, -54.2806, -53.7833, -53.1667,
-51.7333, -51.7, -51.65, -51.6233, -51.5333, -50.3333, -49.983, -49.3, -48.7667, -48.4683, -47.75, -47.2547, -46.9,
-46.795, -46.5886, -46.55, -46.4333, -46.4131, -46.1833, -45.9333, -45.875, -45.8742, -45.8647, -45.6869, -45.6, -4
5.5667, -45.4167, -45.4, -45.0167, -44.865, -44.7, -44.3931, -44.0333, -43.9514, -43.5833, -43.531, -43.386, -43.3,
-43.25, -43.0992, -42.9769, -42.9, -42.8806, -42.7667, -42.7156, -42.6219, -42.4667, -42.45, -42.4, -42.3833, -42.
3, -42.2667, -42.0667, -42.05, -41.9667, -41.8667, -41.7667, -41.7581, -41.6167, -41.5167, -41.4667, -41.4419, -41.
4, -41.3333, -41.3167, -41.3, -41.2889, -41.2708, -41.2581, -41.2167, -41.18, -41.1667, -41.15, -41.1333, -41.1258,
-41.1228, -41.0636, -41.0333, -40.97, -40.9667, -40.9167, -40.8938, -40.875, -40.844, -40.8, -40.7833, -40.7625, -4
0.7333, -40.6219, -40.5833, -40.5725, -40.4, -40.355, -40.3167, -40.2833, -40.2167, -40.1667, -40.1333, -40.1, -40.
0667, -39.9325, -39.9167, -39.8829, -39.85, -39.8139, -39.6667, -39.6444, -39.5933, -39.5, -39.4903, -39.45, -39.36
67, -39.3333, -39.2767, -39.2667, -39.2166, -39.1786, -39.1, -39.0578, -39.0333, -38.9949, -38.9908, -38.9889, -38.
9833, -38.9667, -38.9558, -38.9525, -38.95, -38.9338, -38.9333, -38.9167, -38.9, -38.88, -38.8167, -38.8, -38.7667,
```



The number of comparisons needed to sort the dataset before randomly order the list.

```
print("Number of comparisons, 'before' randomly order latitude data list:", comparison_count_result_random_pivot[0])
```

```
Number of comparisons, 'before' randomly order latitude data list: 627034
```

The number of comparisons needed to sort the dataset after randomly order the list.

```
print("Number of comparisons 'after' deployed randomly reoder latitude dataset:", comparison_count_result_random_dataset[0])
```

```
Number of comparisons 'after' deployed randomly reoder latitude dataset: 678822
```

Analysis:

Before randomizing the dataset, the required number of comparisons for sorting was 627,034. However, after introducing randomness, this number increased to 678,882.

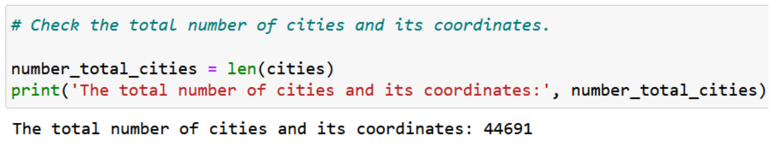
The change in the number of comparisons comes from the random reordering of latitude data prior to applying quicksort. The pivot's ability to divide the arrays determines how efficient the quicksort algorithm is. When data is randomized, a new sequence is introduced with the final element selected as the pivot. As a result, the number of comparisons done throughout the sorting process varies. When the pivot is close to the median, quicksort performs best, providing well-balanced splits with fewer comparisons. An uneven split, on the other hand, may result in additional comparisons if the pivot is not near the median.

In conclusion, the initial order of elements significantly influences the quicksort process's performance.. This randomness disrupts the natural data distribution, leading to increased comparisons and additional work for the quicksort algorithm.

c) Implement a proper quick sort algorithm so that the (latitude, and longitude) pairs are in an ordered list. Use the latitude and longitude values for each city.

Decision-Making in Handling and Sorting (Latitude and Longitude) Pairs:

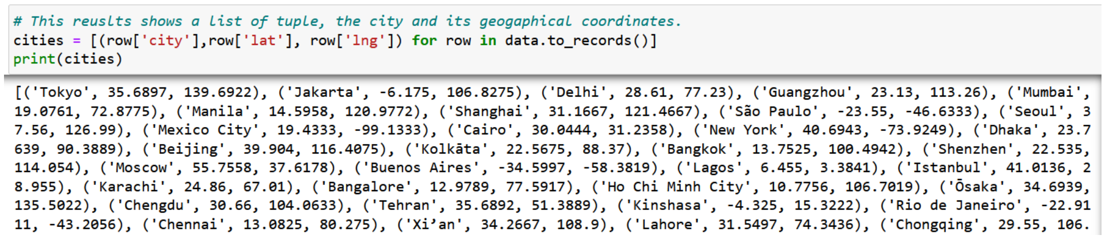
<ins>Dealing with Duplicate Data:</ins> As addressed in question 1 (part c), we opted to include cities with duplicate names which results in a total of 44,691 (latitude and longitude) pairs. This decision was made to ensure the integrity of the dataset.



```
# Check the total number of cities and its coordinates.

number_total_cities = len(cities)
print('The total number of cities and its coordinates:', number_total_cities)
```

```
The total number of cities and its coordinates: 44691
```

<ins>Data Preparation:</ins> To facilitate processing, we transformed the data list into a collection of tuples where each tuple encompasses the city name, latitude, and longitude.



```
# This reuslts shows a list of tuple, the city and its geogaphical coordinates.
cities = [(row['city'],row['lat'], row['lng']) for row in data.to_records()]
print(cities)
```

```
[('Tokyo', 35.6897, 139.6922), ('Jakarta', -6.175, 106.8275), ('Delhi', 28.61, 77.23), ('Guangzhou', 23.13, 113.26), ('Mumbai',
19.0761, 72.8775), ('Manila', 14.5958, 120.9772), ('Shanghai', 31.1667, 121.4667), ('São Paulo', -23.55, -46.6333), ('Seoul', 3
7.56, 126.99), ('Mexico City', 19.4333, -99.1333), ('Cairo', 30.0444, 31.2358), ('New York', 40.6943, -73.9249), ('Dhaka', 23.7
639, 90.3889), ('Beijing', 39.904, 116.4075), ('Kolkāta', 22.5675, 88.37), ('Bangkok', 13.7525, 100.4942), ('Shenzhen', 22.535,
114.054), ('Moscow', 55.7558, 37.6178), ('Buenos Aires', -34.5997, -58.3819), ('Lagos', 6.455, 3.3841), ('Istanbul', 41.0136, 2
8.955), ('Karachi', 24.86, 67.01), ('Bangalore', 12.9789, 77.5917), ('Ho Chi Minh City', 10.7756, 106.7019), ('Ōsaka', 34.6939,
135.5022), ('Chengdu', 30.66, 104.0633), ('Tehran', 35.6892, 51.3889), ('Kinshasa', -4.325, 15.3222), ('Rio de Janeiro', -22.91
11, -43.2056), ('Chennai', 13.0825, 80.275), ('Xi’an', 34.2667, 108.9), ('Lahore', 31.5497, 74.3436), ('Chongqing', 29.55, 106.
```

<ins>Data Sorting Approach:</ins>

To sort the data, we implemented sequential sorting based on latitude, followed by longitude. If the latitudes are equal, we will then compare the longitudes. To explore potential differences in sorting performance, we employed three distinct strategies:

1) Using a Random Pivot Value:
2) Using the last element of the index as a pivot value.
3) Iterative Stack Quicksort with Last Element Pivot.

1) Using a Random Pivot Value:

This code on the image utilizes quicksort for sequential sorting of cities based on latitude and longitude pairs showcasing the effectiveness of quicksort with a random pivot.

Here's a overview:

Quicksort Function (quicksort_lat_lng):

- Performs sequential sorting using a random pivot strategy and then it ecursively sorts the left partition after the pivot.

Partition Function (partition_lat_lng):

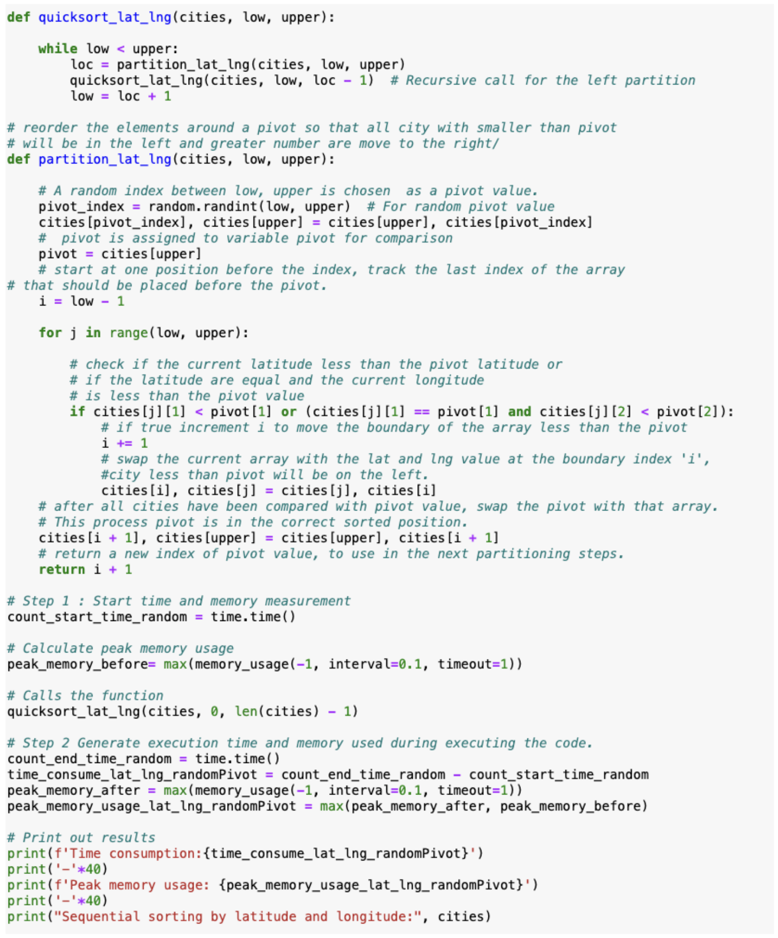
- Reorders elements around a random pivot based on latitude and longitude.

Execution and Measurement:

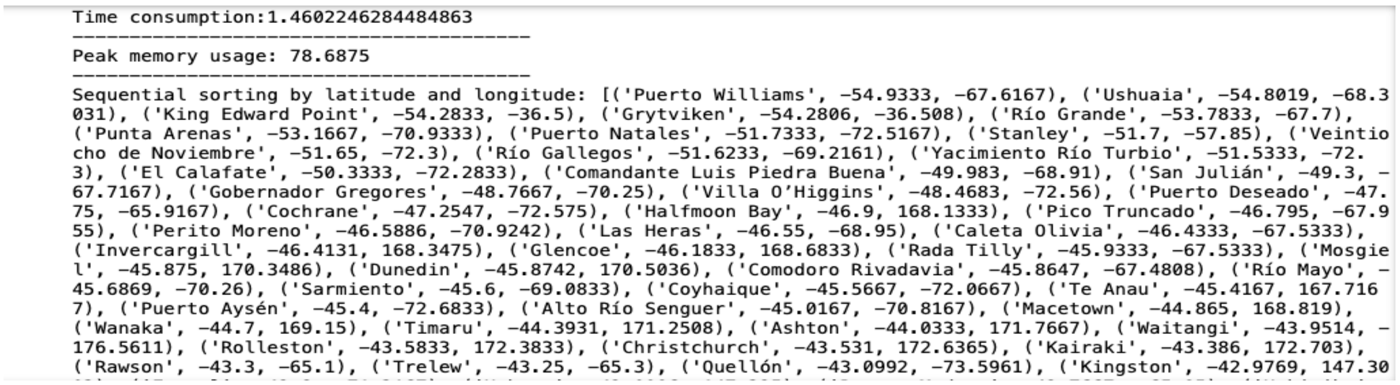
- Measures time and peak memory usage before and after sorting.

Results Printing:

- Prints time consumption, peak memory usage, and the sorted array.



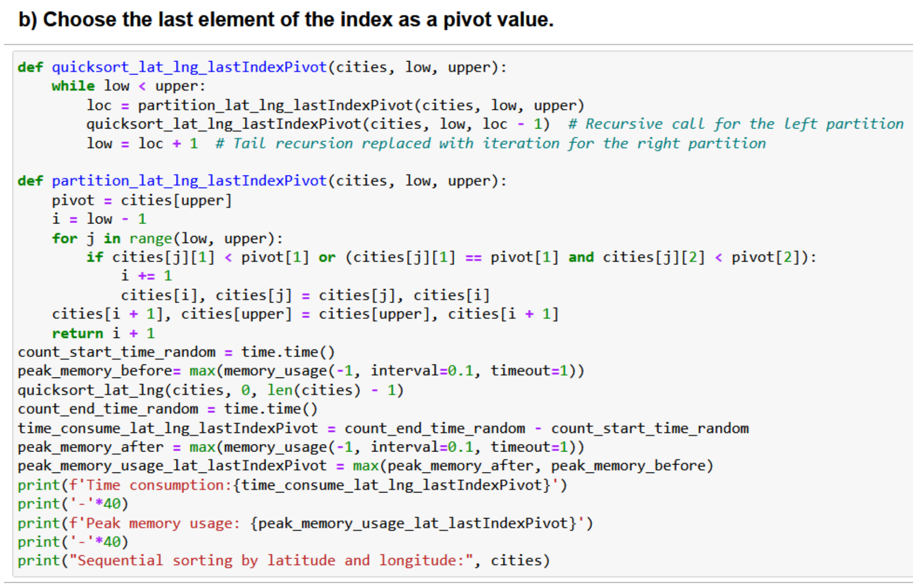
```python
def quicksort_lat_lng(cities, low, upper):

    while low < upper:
        loc = partition_lat_lng(cities, low, upper)
        quicksort_lat_lng(cities, low, loc - 1)  # Recursive call for the left partition
        low = loc + 1

# reorder the elements around a pivot so that all city with smaller than pivot
# will be in the left and greater number are move to the right/
def partition_lat_lng(cities, low, upper):

    # A random index between low, upper is chosen  as a pivot value.
    pivot_index = random.randint(low, upper)  # For random pivot value
    cities[pivot_index], cities[upper] = cities[upper], cities[pivot_index]
    #  pivot is assigned to variable pivot for comparison
    pivot = cities[upper]
    # start at one position before the index, track the last index of the array
# that should be placed before the pivot.
    i = low - 1

    for j in range(low, upper):

        # check if the current latitude less than the pivot latitude or
        # if the latitude are equal and the current longitude
        # is less than the pivot value
        if cities[j][1] < pivot[1] or (cities[j][1] == pivot[1] and cities[j][2] < pivot[2]):
            # if true increment i to move the boundary of the array less than the pivot
            i += 1
            # swap the current array with the lat and lng value at the boundary index 'i',
            #city less than pivot will be on the left.
            cities[i], cities[j] = cities[j], cities[i]
    # after all cities have been compared with pivot value, swap the pivot with that array.
    # This process pivot is in the correct sorted position.
    cities[i + 1], cities[upper] = cities[upper], cities[i + 1]
    # return a new index of pivot value, to use in the next partitioning steps.
    return i + 1

# Step 1 : Start time and memory measurement
count_start_time_random = time.time()

# Calculate peak memory usage
peak_memory_before= max(memory_usage(-1, interval=0.1, timeout=1))

# Calls the function
quicksort_lat_lng(cities, 0, len(cities) - 1)

# Step 2 Generate execution time and memory used during executing the code.
count_end_time_random = time.time()
time_consume_lat_lng_randomPivot = count_end_time_random - count_start_time_random
peak_memory_after = max(memory_usage(-1, interval=0.1, timeout=1))
peak_memory_usage_lat_lng_randomPivot = max(peak_memory_after, peak_memory_before)

# Print out results
print(f'Time consumption:{time_consume_lat_lng_randomPivot}')
print('-'*40)
print(f'Peak memory usage: {peak_memory_usage_lat_lng_randomPivot}')
print('-'*40)
print("Sequential sorting by latitude and longitude:", cities)
```



```
Time consumption:1.4602246284484863
----------------------------------------
Peak memory usage: 78.6875
----------------------------------------
Sequential sorting by latitude and longitude: [('Puerto Williams', -54.9333, -67.6167), ('Ushuaia', -54.8019, -68.3
031), ('King Edward Point', -54.2833, -36.5), ('Grytviken', -54.2806, -36.508), ('Río Grande', -53.7833, -67.7),
('Punta Arenas', -53.1667, -70.9333), ('Puerto Natales', -51.7333, -72.5167), ('Stanley', -51.7, -57.85), ('Veintio
cho de Noviembre', -51.65, -72.3), ('Río Gallegos', -51.6233, -69.2161), ('Yacimiento Río Turbio', -51.5333, -72.
3), ('El Calafate', -50.3333, -72.2833), ('Comandante Luis Piedra Buena', -49.983, -68.91), ('San Julián', -49.3, -
67.7167), ('Gobernador Gregores', -48.7667, -70.25), ('Villa O’Higgins', -48.4683, -72.56), ('Puerto Deseado', -47.
75, -65.9167), ('Cochrane', -47.2547, -72.575), ('Halfmoon Bay', -46.9, 168.1333), ('Pico Truncado', -46.795, -67.9
55), ('Perito Moreno', -46.5886, -70.9242), ('Las Heras', -46.55, -68.95), ('Caleta Olivia', -46.4333, -67.5333),
('Invercargill', -46.4131, 168.3475), ('Glencoe', -46.1833, 168.6833), ('Rada Tilly', -45.9333, -67.5333), ('Mosgie
l', -45.875, 170.3486), ('Dunedin', -45.8742, 170.5036), ('Comodoro Rivadavia', -45.8647, -67.4808), ('Río Mayo', -
45.6869, -70.26), ('Sarmiento', -45.6, -69.0833), ('Coyhaique', -45.5667, -72.0667), ('Te Anau', -45.4167, 167.716
7), ('Puerto Aysén', -45.4, -72.6833), ('Alto Río Senguer', -45.0167, -70.8167), ('Macetown', -44.865, 168.819),
('Wanaka', -44.7, 169.15), ('Timaru', -44.3931, 171.2508), ('Ashton', -44.0333, 171.7667), ('Waitangi', -43.9514, -
176.5611), ('Rolleston', -43.5833, 172.3833), ('Christchurch', -43.531, 172.6365), ('Kairaki', -43.386, 172.703),
('Rawson', -43.3, -65.1), ('Trelew', -43.25, -65.3), ('Quellón', -43.0992, -73.5961), ('Kingston', -42.9769, 147.30
```

2) Using the last element of the index as a pivot value.

This code in the image shows how quicksort sequentially sorts cities based on latitude and longitude, using the last element as the pivot. Here's a concise breakdown:



b) Choose the last element of the index as a pivot value.

```python
def quicksort_lat_lng_lastIndexPivot(cities, low, upper):
    while low < upper:
        loc = partition_lat_lng_lastIndexPivot(cities, low, upper)
        quicksort_lat_lng_lastIndexPivot(cities, low, loc - 1)  # Recursive call for the left partition
        low = loc + 1  # Tail recursion replaced with iteration for the right partition

def partition_lat_lng_lastIndexPivot(cities, low, upper):
    pivot = cities[upper]
    i = low - 1
    for j in range(low, upper):
        if cities[j][1] < pivot[1] or (cities[j][1] == pivot[1] and cities[j][2] < pivot[2]):
            i += 1
            cities[i], cities[j] = cities[j], cities[i]
    cities[i + 1], cities[upper] = cities[upper], cities[i + 1]
    return i + 1
count_start_time_random = time.time()
peak_memory_before= max(memory_usage(-1, interval=0.1, timeout=1))
quicksort_lat_lng(cities, 0, len(cities) - 1)
count_end_time_random = time.time()
time_consume_lat_lng_lastIndexPivot = count_end_time_random - count_start_time_random
peak_memory_after = max(memory_usage(-1, interval=0.1, timeout=1))
peak_memory_usage_lat_lastIndexPivot = max(peak_memory_after, peak_memory_before)
print(f'Time consumption:{time_consume_lat_lng_lastIndexPivot}')
print('-'*40)
print(f'Peak memory usage: {peak_memory_usage_lat_lastIndexPivot}')
print('-'*40)
print("Sequential sorting by latitude and longitude:", cities)
```

Quicksort Function (quicksort_lat_lng_lastIndexPivot):

- Sequentially sorts cities using the last element as a pivot and then it recursively sorts the left partition after the pivot.

Partition Function (partition_lat_lng_lastIndexPivot):

- Uses the last element as the pivot for reordering and then compares latitude and longitude to determine the order.

Execution and Measurement:

- Measures time and peak memory usage before and after sorting.

Results Printing:

- Prints time consumption, peak memory usage, and the sorted array.

Analysis three different way of choosing pivot value:

Analyzing Three Pivot Selection Strategies:

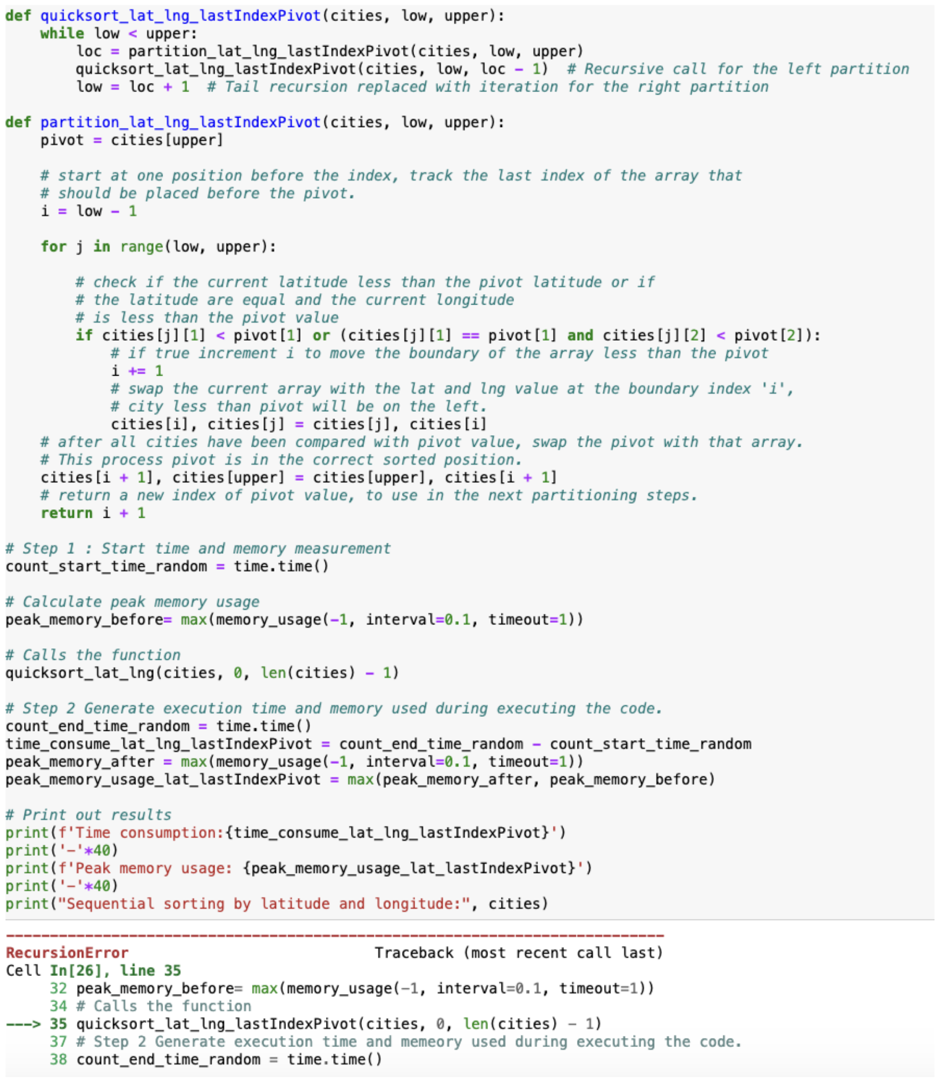
1. Random Pivot Quicksort:

Successfully sorts latitude and longitude data in ascending order. Additionally, it is fast and the memory is efficient due to a balanced partition process. It utilizes a random pivot to enhance the expected time complexity to O(N log N). The algorithm generally performs well even if the worst-case complexity remains the same O(n^2).

2. Last Element Pivot Quicksort:

After running the algorithm for 5 minutes, a 'RecursionError' occurred during the implementation of quicksort, specifically 'maximum recursion depth exceeded in comparison.' This error indicates that the recursive became too deep while sorting pairs of latitude and longitude which resulted in a Python interpreter limit or a 'stack overflow.'

Such issues in quicksort often arise when the chosen pivot, in this case, the last element, isn't well-suited to efficiently, reducing the array size during the recursive calls. This can lead to worst-case performance of O(n^2) (Stack Overflow. (n.d.), 2023), as the depth of recursive calls increases linearly with the array size, reflecting a highly unbalanced proportion.



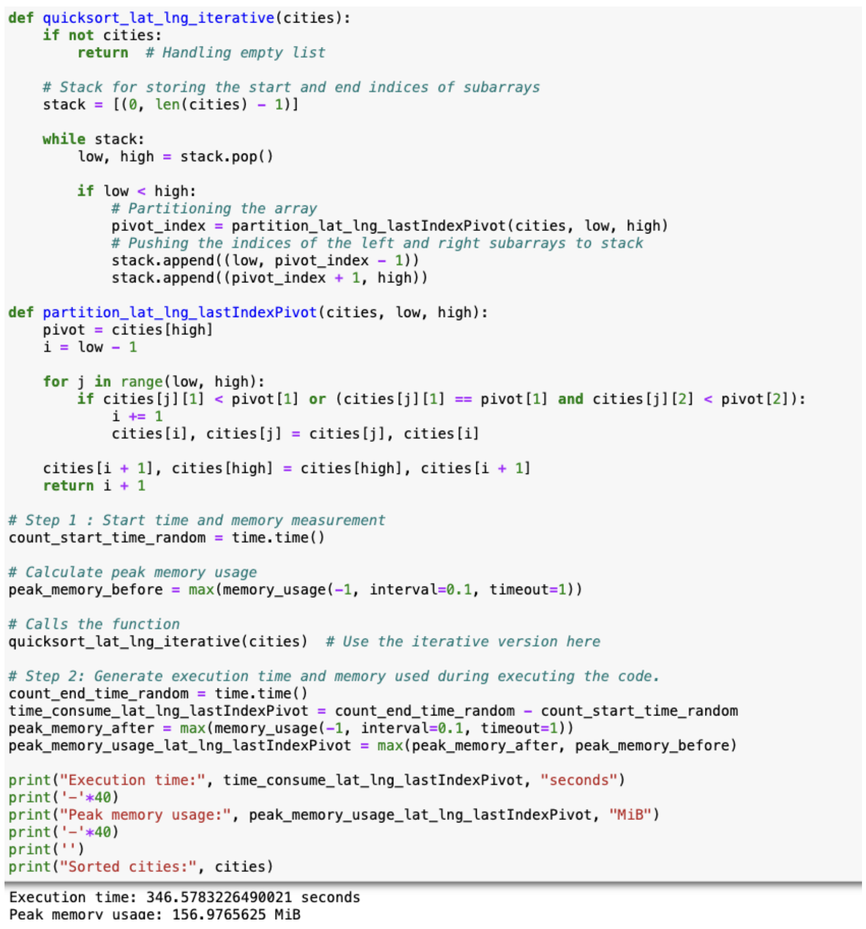
```
def quicksort_lat_lng_lastIndexPivot(cities, low, upper):
    while low < upper:
        loc = partition_lat_lng_lastIndexPivot(cities, low, upper)
        quicksort_lat_lng_lastIndexPivot(cities, low, loc - 1)  # Recursive call for the left partition
        low = loc + 1  # Tail recursion replaced with iteration for the right partition

def partition_lat_lng_lastIndexPivot(cities, low, upper):
    pivot = cities[upper]

    # start at one position before the index, track the last index of the array that
    # should be placed before the pivot.
    i = low - 1

    for j in range(low, upper):

        # check if the current latitude less than the pivot latitude or if
        # the latitude are equal and the current longitude
        # is less than the pivot value
        if cities[j][1] < pivot[1] or (cities[j][1] == pivot[1] and cities[j][2] < pivot[2]):
            # if true increment i to move the boundary of the array less than the pivot
            i += 1
            # swap the current array with the lat and lng value at the boundary index 'i',
            # city less than pivot will be on the left.
            cities[i], cities[j] = cities[j], cities[i]
    # after all cities have been compared with pivot value, swap the pivot with that array.
    # This process pivot is in the correct sorted position.
    cities[i + 1], cities[upper] = cities[upper], cities[i + 1]
    # return a new index of pivot value, to use in the next partitioning steps.
    return i + 1

# Step 1 : Start time and memory measurement
count_start_time_random = time.time()

# Calculate peak memory usage
peak_memory_before= max(memory_usage(-1, interval=0.1, timeout=1))

# Calls the function
quicksort_lat_lng(cities, 0, len(cities) - 1)

# Step 2 Generate execution time and memory used during executing the code.
count_end_time_random = time.time()
time_consume_lat_lng_lastIndexPivot = count_end_time_random - count_start_time_random
peak_memory_after = max(memory_usage(-1, interval=0.1, timeout=1))
peak_memory_usage_lat_lastIndexPivot = max(peak_memory_after, peak_memory_before)

# Print out results
print(f'Time consumption:{time_consume_lat_lng_lastIndexPivot}')
print('-'*40)
print(f'Peak memory usage: {peak_memory_usage_lat_lastIndexPivot}')
print('-'*40)
print("Sequential sorting by latitude and longitude:", cities)
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
Cell In[26], line 35
     32 peak_memory_before= max(memory_usage(-1, interval=0.1, timeout=1))
     34 # Calls the function
---> 35 quicksort_lat_lng_lastIndexPivot(cities, 0, len(cities) - 1)
     37 # Step 2 Generate execution time and memeory used during executing the code.
     38 count_end_time_random = time.time()
```

To address this, further will be needed, where alternative algorithms can be explored by changing the pivot selection method to random, mean, or median which can help to enhance average performance and reduce issues that are associated with unbalanced recursive calls. Additionally, if we are lucky and partition size are balanced, this code can efficiently show fast execution in small proportion of memory usage, however it will depend on the environment in which the algorithm is executed for example ongoing programs that might affects the run time.

3. Iterative Stack Quicksort with Last Element Pivot:

Encountered a 'RecursionError' in the second approach, where the last element was used as a pivot this made us explore alternative algorithms. After employing an iterative stack-based approach, adapted from online resources (GeeksforGeeks, 2012), we were able to successfully address the 'RecursionError' and produce accurate sorting of the dataset in sequential order by latitude and longitude. The execution time was approximately 500 seconds, with a million recorded comparisons, while peak memory usage remained comparable to other methods.



```python
def quicksort_lat_lng_iterative(cities):
    if not cities:
        return  # Handling empty list

    # Stack for storing the start and end indices of subarrays
    stack = [(0, len(cities) - 1)]

    while stack:
        low, high = stack.pop()

        if low < high:
            # Partitioning the array
            pivot_index = partition_lat_lng_lastIndexPivot(cities, low, high)
            # Pushing the indices of the left and right subarrays to stack
            stack.append((low, pivot_index - 1))
            stack.append((pivot_index + 1, high))

def partition_lat_lng_lastIndexPivot(cities, low, high):
    pivot = cities[high]
    i = low - 1

    for j in range(low, high):
        if cities[j][1] < pivot[1] or (cities[j][1] == pivot[1] and cities[j][2] < pivot[2]):
            i += 1
            cities[i], cities[j] = cities[j], cities[i]

    cities[i + 1], cities[high] = cities[high], cities[i + 1]
    return i + 1

# Step 1 : Start time and memory measurement
count_start_time_random = time.time()

# Calculate peak memory usage
peak_memory_before = max(memory_usage(-1, interval=0.1, timeout=1))

# Calls the function
quicksort_lat_lng_iterative(cities)  # Use the iterative version here

# Step 2: Generate execution time and memory used during executing the code.
count_end_time_random = time.time()
time_consume_lat_lng_lastIndexPivot = count_end_time_random - count_start_time_random
peak_memory_after = max(memory_usage(-1, interval=0.1, timeout=1))
peak_memory_usage_lat_lng_lastIndexPivot = max(peak_memory_after, peak_memory_before)

print("Execution time:", time_consume_lat_lng_lastIndexPivot, "seconds")
print('-'*40)
print("Peak memory usage:", peak_memory_usage_lat_lng_lastIndexPivot, "MiB")
print('-'*40)
print('')
print("Sorted cities:", cities)
```

```
Execution time: 346.5783226490021 seconds
Peak memory usage: 156.9765625 MiB
```

In summary, implementing the quicksort algorithm with different pivot values produces varied outcomes in terms of execution time, memory usage, and the potential for errors. Randomized pivots contribute to more balanced splits and improved performance. However, iterative implementations using a stack, while avoiding 'recursive exceeded' errors, may exhibit suboptimal outcomes as well as longer execution times.

Operation	Execution time (seconds)	Peak memory usage (MiB)
(a) Quick sort process for latitude data (random pivot)	1.32081	73.58593
(a) Quick sort process for latitude data (last element as a pivot)	1.258035	77.6679
(b) Quick sort process for latitude data randomly reordering the list before sorting **(Counting the number of comparisons)**	1.217647	72.91796
(c) quick sort using (Pivot value)	1.460224	78.6875
(c) quick sort using (last element as a pivot)	“Error”	Error”
(c) quick sort coordinates by distance using (last element as a pivot with stack method)	346.5783	156.9765

Overall Observation:

1. **Compare time and memory execution in each merge sort process :**

Good balance when random select pivot value : Results indicate that the segments using random pivot values are more efficient, providing balanced partitions in QuickSort. This is evident in (C-1), where the pivot leads to faster execution and comparable memory usage to basic QuickSort (a and b). Random pivot selection helps avoid imbalanced partitions, as each number has an equal chance of becoming a pivot while promoting a more balanced average partition.

Last element as a pivot : The table shows a slight improvement in time execution when compared to random pivot, while the peak memory usage has increased.

Sorting Latitude and Longitude Pairs:

- Pivot Value Performance:
  - Optimal time execution and memory usage, slightly exceeding standard quicksort for simple latitude data.
- Last Element as Pivot:
  - Errors in one environment, successful in another, indicating environmental set up and concurrent running programs can cause the issue of execution.
- Stack Element with Last Element Pivot:
  - Significant increases in execution times and memory usage, unsuitable for older latitude and longitude data lists.

Complexity Analysis of Quick Sort after Sorting Latitude and (Latitude, Longitude) Pair Data:

- Complexity:
  - Quick sort's time complexity scales linearly with individual elements (n).

In Conclusion the pivot selection significantly influences quicksort's performance, particularly with large datasets. Therefor, strategic pivot choices are crucial for best outcomes.

VI. Comparison of MergeSort and QuickSort:

Both mergesort and quicksort use the basic world cities database, and employ the same latitude values and (latitude, longitude) pairs. Here are the key findings and differences between these algorithms:

Sorting Unique Latitude Values:

MergeSort 1(a) and 1(b):

- The latitude dataset is successfully sorted in ascending order using MergeSort.
- Randomization of Latitude Dataset: Testing the algorithm's adaptability by randomizing the order of latitude values to observe any changes.
  - Number of Merge Operations:

- The number of merge operations in MergeSort remains consistent. MergeSort divides and merges back to a single element, with the algorithm's efficiency depending on the dataset's size rather than its initial order.
    - Number of Comparisons during MergeSort:

The number of comparisons constantly changes after code execution. These individual evaluations determine the hierarchical order of components in two subarrays which results in different comparison numbers based on the array's original order.

Quick sort 2 (a) and 2(b):

QuickSort performs well when sorting city latitudes, exhibiting superior sorting performance. This is shown in lower peak memory usage and fast execution times. while slightly slower than merge sort. However, Quick sort can perform more effectively especially when considering variations in pivot selection.

Random Pivot Quicksort:Efficiently sorts city latitude with faster execution time and less peak memory usage.

Last Index as a Pivot Element:

- Successfully sorts city latitude with a similar time and peak memory profile as a random pivot quicksort.

Number of Comparisons in a Randomized Order List:

- The number of comparisons varies with each execution. Unlike MergeSort, QuickSort sorts data based on the position of each pivot element. The pre-sorted data can lead to unbalanced partitions which results in a higher number of comparisons.

Sorting latitude and longitude pairs.

MergeSort Issue 1 (C):

The implementation efficiently sorts latitude and longitude pairs. When employing the Haversine Formula for distance-based sorting, the time remains consistent but there is a slightly higher peak memory consumption. This may be because of the computational demands during sorting or the size of the dataset.

Second approach: Sequential Sorting by latitude followed by longitude proves efficient, maintaining the same criteria order of the elements.

QuickSort Issue 2 (C):

Similar to MergeSort, QuickSort performs well when sorting by latitude first and then longitude. However, the result is significantly influenced by pivot selection.

- Using a random pivot value produces the best result in quick execution time. While, choosing the last element of the index as a pivot value leads to a recursion error.
- Iterative stack quicksort, which uses the last element as a pivot, provides accurate results but requires a longer execution time.

VII. Conclusion and future work

In summary, the exploration of Merge Sort and QuickSort algorithms for sorting geographical coordinates, including latitude data and (latitude, longitude) pairs, provided valuable insights into their efficiency and adaptability.

Merge Sort:

- Demonstrated reliable performance with O(n log n) time and space complexity.
- Robust handling of diverse scenarios, regardless of initial dataset order.
- Sequential sorting by latitude and longitude showcased comparable efficiency.

QuickSort:

- Pivot selection significantly influenced sorting efficiency.
- Random pivot selection resulted in faster execution times and lower memory usage.
- Sensitivity to the initial order of elements was observed, specially when randomizing the dataset.

With their own set of considerations, Merge Sort and QuickSort both demonstrated their adaptability and efficiency. Merge Sort handled latitude data well, exhibiting consistent performance and flexibility. QuickSort showed agility, with pivot selection being an important factor. These research contribute to our knowledge of effective sorting in many geographic coordinate circumstances.

VIII. References

Stack Overflow. (n.d.). *Quicksort: Choosing the pivot*. [online] Available at: https://stackoverflow.com/questions/164163/quicksort-choosing-the-pivot

Stack Overflow. (n.d.). *Getting distance between two points based on latitude/longitude*. [online] Available at: https://stackoverflow.com/questions/19412462/getting-distance-between-two-points-based-on-latitude-longitude

Medium. (n.d.). *Medium*. [online] Available at: https://louwersj.medium.com/calculate-geographic-distances-in-python-with-the-haversine-method-ed99b41ff04b

Khan Academy. (n.d.). *Challenge: Implement quicksort | Quick sort | Algorithms | Computer science theory | Computing*. [online] Available at: https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/pc/challenge-implement-quicksort

GeeksforGeeks. (2012). *Iterative Quick Sort*. [online] Available at: https://www.geeksforgeeks.org/iterative-quick-sort/

Gautam, S. (n.d.). *Merge Sort Algorithm*. [online] www.enjoyalgorithms.com. Available at: https://www.enjoyalgorithms.com/blog/merge-sort-algorithm

Naif Aljabri, Muhammad Al-Hashimi, Saleh, M. and Osama Ahmed Abulnaja (2019). Investigating power efficiency of mergesort. *The Journal of Supercomputing*, 75(10), pp.6277–6302. doi:https://doi.org/10.1007/s11227-019-02850-5

Stack Overflow. (n.d.). *Python QuickSort maximum recursion depth*. [online] Available at: https://stackoverflow.com/questions/27116255/python-quicksort-maximum-recursion-depth

GeeksforGeeks (2018). *Merge Sort - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/merge-sort/

Stack Overflow. (n.d.). *Array Merge sort Sorting Count and Sorting time Python*. [online] Available at: https://stackoverflow.com/questions/67534291/array-merge-sort-sorting-count-and-sorting-time-python

www.youtube.com. (n.d.). *7.7 Merge Sort in Data Structure | Sorting Algorithms| DSA Full Course*. [online] Available at: https://www.youtube.com/watch?v=jlHkDBEumP0

davidkmw0810 (2018). argorithm/Foundations of Algorithms - Richard E. Neapolitan.pdf at master · davidkmw0810/argorithm. [online] GitHub. Available at: https://github.com/davidkmw0810/argorithm/blob/master/Foundations%20of%20Algorithms%20-%20Richard%20E.%20Neapolitan.pdf