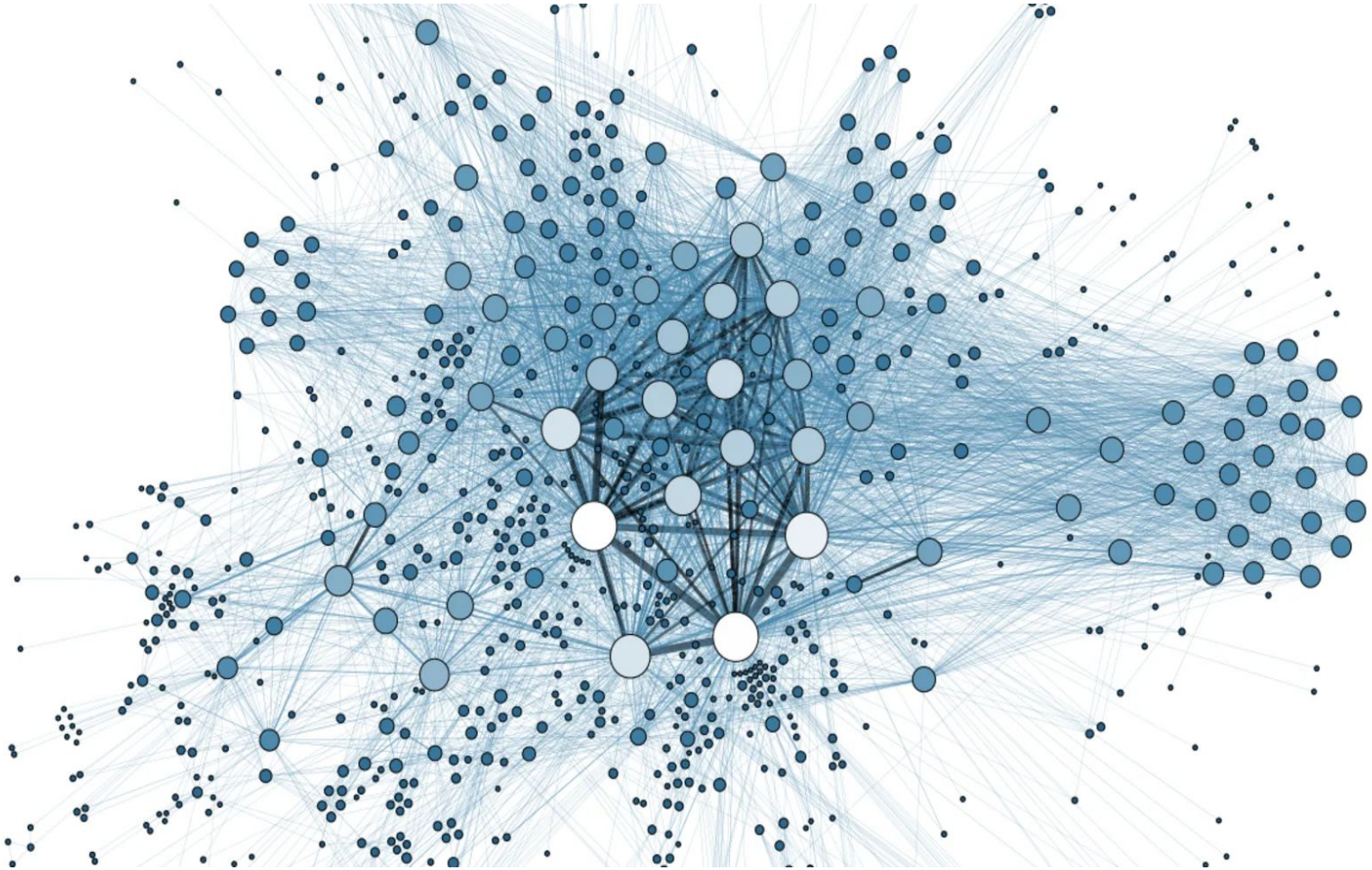
Kristiania

Analysis of Merge Sort & Quicksort Algorithms in Geographical Data



Agenda

- Introduction
- Data Source
- Environment setup
- Methodology
  - 1) Data Preprocessing
- Results and Discussion
  - **Merge Sort Problem**
  - **Quick sort Problem**
- Overall observation
- Conclusion

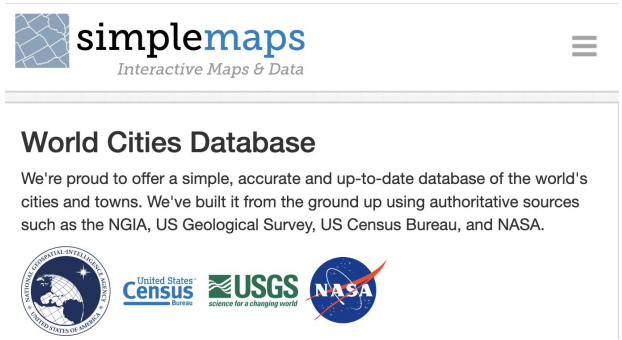
Data: **The Basic World Cities Dataset**

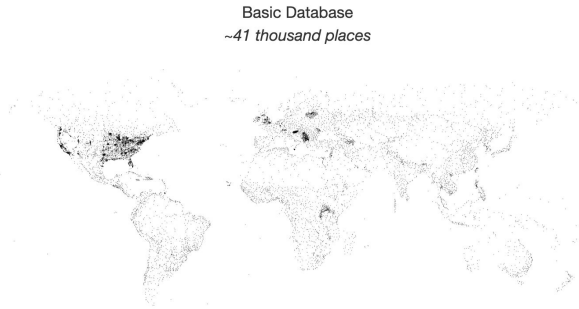
Source : **https://simplemaps.com/data/world-cities**

Total number of values: **44,691**

```
data = pd.read_csv('worldcities.csv')
data.head()
```

	city	city_ascii	lat	lng	country	iso2	iso3	admin_name	capital
0	Tokyo	Tokyo	35.6897	139.6922	Japan	JP	JPN	Tōkyō	primary
1	Jakarta	Jakarta	-6.1750	106.8275	Indonesia	ID	IDN	Jakarta	primary
2	Delhi	Delhi	28.6100	77.2300	India	IN	IND	Delhi	admin
3	Guangzhou	Guangzhou	23.1300	113.2600	China	CN	CHN	Guangdong	admin
4	Mumbai	Mumbai	19.0761	72.8775	India	IN	IND	Mahārāshtra	admin





Agenda

- Introduction
- Data Source
- Environment setup

- **Python 3.11.4**
- **Jupyter Notebook**
- **Monitor time and peak memory**
  - Load Time Library
  - Load Memory_profiler Libary

Work with Python version: 3.11.4 | Jupyter Notebook

```
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)]
```

Monitor time and memory usgae

```
# 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('Execution time:', time_consume_randomPivot, 'seconds')
```

Agenda

- Introduction
- Data Source
- Environment Setup
- Methodology
  - 1) Data Preprocessing

Latitude

Step 1: Download the data

```
data = pd.read_csv('worldcities.csv')
data.head()
```

	city	city_ascii	lat	lng	country	iso2	iso3	admin_name	capital	population	id
0	Tokyo	Tokyo	35.6897	139.6922	Japan	JP	JPN	Tōkyō	primary	37732000.0	1392685764
1	Jakarta	Jakarta	-6.1750	106.8275	Indonesia	ID	IDN	Jakarta	primary	33756000.0	1360771077
2	Delhi	Delhi	28.6100	77.2300	India	IN	IND	Delhi	admin	32226000.0	1356872604
3	Guangzhou	Guangzhou	23.1300	113.2600	China	CN	CHN	Guangdong	admin	26940000.0	1156237133
4	Mumbai	Mumbai	19.0761	72.8775	India	IN	IND	Mahārāshtra	admin	24973000.0	1356226629

Step 2: Select only latitude values

```
# select only latitude column
df = data['lat']
print(f'List of latitude values : {df.head()}')
print('')
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
```

Step 3: Check for duplication and elimination

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

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

Step 4: Numpy array and Final number for sorting

```
print(f'Shape : {dfn.shape}')
```

```
Types of Latitude values : <class 'numpy.ndarray'>
--------------------------------------------------
Latitude values in one dimentional numpy array form : [ 35.6897  -6.175
--------------------------------------------------
Shape : (34657,)
```

Latitude and Longitude

Step 1: Download the data

```
data = pd.read_csv('worldcities.csv')
data.head()
```

	city	city_ascii	lat	lng	country	iso2	iso3	admin_name	capital	population	id
0	Tokyo	Tokyo	35.6897	139.6922	Japan	JP	JPN	Tōkyō	primary	37732000.0	1392685764
1	Jakarta	Jakarta	-6.1750	106.8275	Indonesia	ID	IDN	Jakarta	primary	33756000.0	1360771077
2	Delhi	Delhi	28.6100	77.2300	India	IN	IND	Delhi	admin	32226000.0	1356872604
3	Guangzhou	Guangzhou	23.1300	113.2600	China	CN	CHN	Guangdong	admin	26940000.0	1156237133
4	Mumbai	Mumbai	19.0761	72.8775	India	IN	IND	Mahārāshtra	admin	24973000.0	1356226629

Step 2: Select only latitude and longitude values

```
# 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(pair_cities_la_and_long[30])
```

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

Step 3: Check for uplication values

Duplication Verification for Identical City Names and Geographical Coordinates:

```
# 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
```

Step 4: List of tuple and Final number for sorting

```
# 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
```

Agenda

- Introduction
- Data Source
- Environment Setup
- Methodology
  - 1) Data Preprocessing
- Results and Discussion
  - **Merge Sort Problem**
  - **Quick sort Problem**

Problem 1: Merge Sort

What is merge sort?

Pseudo Code

- Task 1
- Task 2
- Task 3

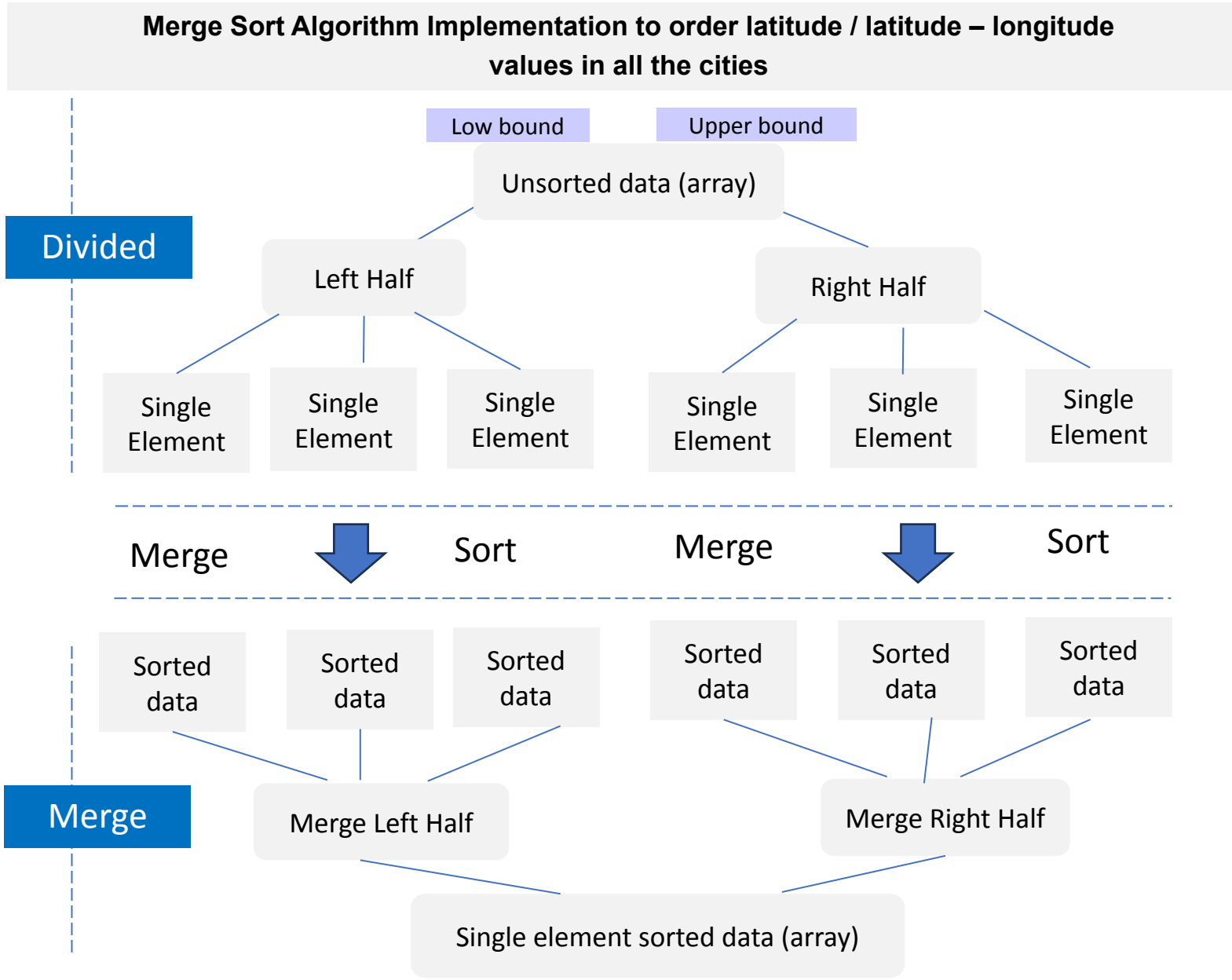
Conclusion

Merge Sort

What is merge sort?

Pseudocode
- Part a
- Part b
- Part c

Conclusion



Merge Sort

What is merge sort?

Pseudocode
- Part a
- Part b
- Part c

Conclusion

Pseudo code

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

Merge Sort

What is merge sort?

Pseudocode

- Part a
- Part b
- Part c

Conclusion

Main code for Merge sort Process

```
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 = 0
    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))
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 for latitude merge sort', time_consume_and_counting_number_merge_operation, 'seconds')
print('Memory usgae', peak_memory_usage_and_counting_number_merge_operation, 'MiB')
print('Total number of merges sort during merge operation:', count_number_of_merge)
print('The output after merge sorted is:\n\n', sorted_number)
```

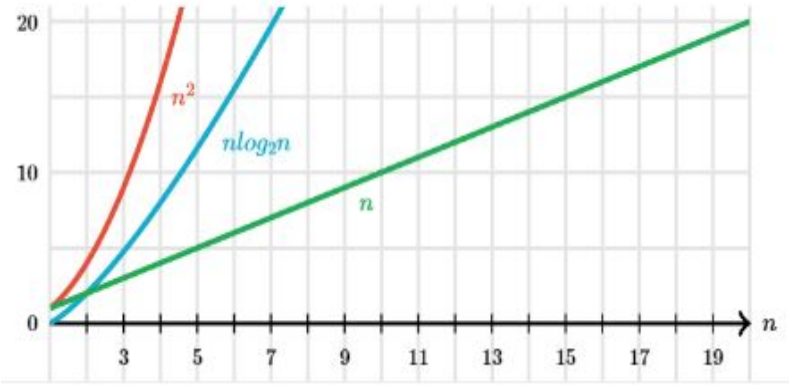
Merge Sort

What is merge sort?

Pseudocode

- Part a
- Part b
- Part c

Conclusion



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

I. Merging process

Step 1: Divide and Recursive

- **Initial Step:** Divide the array into two halves.
- **Bounds:** Set lower bound to 0, upper bound to 34,657.
- **Mid-Point Calculation:** Approx. 17,273.5 (dividing sum of bounds by 2)

Step 2 : Merge

- First Half: Index 0 to 17,274.
- Second Half: Index 17,275 to 34,657

II. Performance Evaluation

- **Time Complexity:** O(n log n) for 'n' elements.
  - Enhanced by the division process and linear-time merging.
- **Memory Usage:**
  - Requires additional space for merging.
  - Memory efficiency maintained as space is reused

Conclusion:

- Merge sort is effective for sorting a 34,657-element dataset of latitudes.
- Efficiency validated by measured execution time and peak memory within acceptable limits for data size.

```
------------------------------------------------
Total number of merges sort during merge operation: 34656
------------------------------------------------
```

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

```
[-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, -43.5833, -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, -40.7333, -40.6219, -40.5833, -40.5725, -40.4, -40.355, -40.3167, -40.2833, -40.2167, -40.1667, -40.1333, -40.1, -40.0667, -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, -39.2166, -39.1786, -39.1, -39.0578, -39.0333, -38.9949, -38.9908, -38.9889, -38.9833,
```

Merge Sort

What is merge sort?

Pseudocode

- Part a
- Part b
- Part c

Conclusion

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?

I. Count the number of merge

1) Count the number of merge during operation
2) Count the number of comparisons.

II. Does it change if you randomly order the list before sorting?

1) Count the number of merge during operation
2) Count the number of comparisons.

I. Count the number of merge

Approach 1) Count the number of merge during operation

- **Method :** Increment 'count_number_of_merge_operation'
- **Process :** Sort and count the number during sort operation
- **Total :** For 34,657 elements, total merge operations = **34,656** (n-1 for 'n' elements).

```
def mergesort_and_counting_number_merge_operation(arr, low, high):
    if low >= high:
    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_countsc
def merge_and_count(left, right):
    k = []
    count_number_of_merge = 0 # Initialsize count as 0
    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
```

```
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
```

Approach 2) Count the number of Comparison

- **Method :** Increment 'mergesort_and_counting_number_merge_comparisons'.
- **Process :** Sort and count comparisons during merging.
- **Total Comparisons :** 478,842 for the dataset.

```
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)
```

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

II. Does it change if you randomly order the list before sorting?

Kristiania

Method :

- Randomly rearrange dataset.
- Apply Merge Sort and observe changes in merge counts and comparisons.

Finding 1) Count the number of merge

- **No change in total merge count regardless of data order.**
- **Reason:** Merge count depends on the number of elements, not their initial order.
- **Performance:** Consistent in time and space complexity

Randomly reoder the dataset

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

```
[ 40.0667  -4.8311 -35.0167 ...  33.9417  22.28     3.52  ]
```

The number of merge operations remain the same, same as the ordering list.

```
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
```

Finding 2) Count the number of comparisons

- **Number of comparisons varies with data order.**
- **Reason:** Comparisons depend on the relative order of elements in subarrays.
- **Performance:** More comparisons may be needed in randomly ordered arrays, affecting efficiency.

Randomly reoder the dataset

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

```
[ 40.0667  -4.8311 -35.0167 ...  33.9417  22.28     3.52  ]
```

The number of comparions change when apply randomly order dataset, same as the ordering list.

```
print('-'* 38)
print('Total comparisons needed to sort the dataset completely using MergeSort:',
      total_count_number_of_comparisons)
```

```
--------------------------------------
Total comparisons needed to sort the dataset completely using MergeSort: 479210
```

Agenda

What is merge sort?

Pseudocode

- Part a
- Part b
- Part c

Conclusion

Use the latitude and longitude values for each city.

Step 1: Download the data

```
data = pd.read_csv('worldcities.csv')
data.head()
```

	city	city_ascii	lat	lng	country	iso2	iso3	admin_name	capital	population	id
0	Tokyo	Tokyo	35.6897	139.6922	Japan	JP	JPN	Tōkyō	primary	37732000.0	1392685764
1	Jakarta	Jakarta	-6.1750	106.8275	Indonesia	ID	IDN	Jakarta	primary	33756000.0	1360771077
2	Delhi	Delhi	28.6100	77.2300	India	IN	IND	Delhi	admin	32226000.0	1356872604
3	Guangzhou	Guangzhou	23.1300	113.2600	China	CN	CHN	Guangdong	admin	26940000.0	1156237133
4	Mumbai	Mumbai	19.0761	72.8775	India	IN	IND	Mahārāshtra	admin	24973000.0	1356226629

Step 2: Select only latitude values

```
# select only latitude column
df = data['lat']
print(f'List of latitude values : {df.head()}')
print('')
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
```

Step 3: Check for duplication and elimination

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

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

Step 4: Numpy array and Final number for sorting

```
print(f'Shape : {dfn.shape}')
```

```
Types of Latitude values : <class 'numpy.ndarray'>
--------------------------------------------------
Latitude values in one dimentional numpy array form : [ 35.6897  -6.175
--------------------------------------------------
Shape : (34657,)
```

Data : Latitude and Longitude

Type : List of Tuple

Objective: Sort 44,691 pairs of geographic coordinates (latitude, longitude) using Merge Sort.

Variable name : *'pair_cities_la_and_long'*

Decision making : Sort all pairs, including duplicates, for comprehensive geographic organization.

Duplication Handling: 3348 duplicate city names, 137 duplicate coordinate pairs.

Merge Sort

What is merge sort?

Pseudocode

- Part a
- Part b
- Part c

Conclusion

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

	Method 1: Merge Sort by Distance	Method 2: Sequential Sorting
Method	• Utilize Haversine formula for distance calculation. • Order by ascending distance from a fixed point (e.g., Bangkok).	• Sort first by latitude, then by longitude.

Method 1) Merge sort Coordinates by Distance using Havers

Step 3: Testing & Validation

Step 1 : Utilize Haversine formula for distance calculation.

```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
```

Step 2 : Order by ascending distance from a fixed point (e.g., Bangkok).

- **Sorting Criterion:** Ascending order based on distance to Bangkok (lat=13.736717, long=100.523186).

```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)
```

```
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,
```

```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
```

```
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
```

Observation

Performance: Small increase in time and memory usage

Implications: Indicates potential impact of dataset size or computational complexity.

Method 2 Additional exploration : Sequential Sorting by latitude followed by longitude

```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)
```

Exploratory Sorting Method:

- Adopted a secondary sorting approach without distance calculations.
- Sorted data first by latitude, then by longitude for geographical accuracy.

Efficiency Observations:

- Demonstrated improved memory efficiency, with reduced peak memory usage compared to distance-based sorting.
- Slight increase in time execution, attributed to the absence of complex distance computations

```
------------------------------
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',
```

Part C : Results

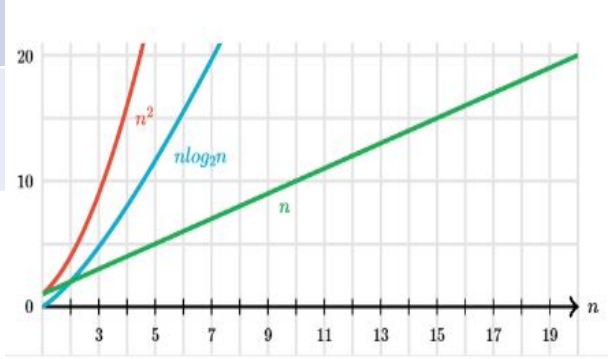
	Method 1: Merge Sort by Distance	Method 2: Sequential Sorting
Method	• Utilize Haversine formula for distance calculation. • Order by ascending distance from a fixed point (e.g., Bangkok)	• Sort first by latitude, then by longitude.
Efficiently	• Slight increase in time and memory usage.	• More efficient in memory usage, slight increase in time

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

Summarize : Overall performance and scalability of the algorithm.

- **Complexity analysis of merge sort.**

Criteria	Part (a) Basic Merge Sort	Part (b) Merge Count	Part (c) With Haversine Formula	Part (c) Sequential Sorting by Latitude & Longitude
Execution Time	Consistent	Consistent	Slighlty higher but consider Consistent	Higher than others
Memory Usage	Consistent	Consistent	Slightly Increased	Slightly lower
Time Complexity	O(n log n)	O(n log n)	O(n log n)	O(n log n)
Space Complexity	O(n)	O(n) (Potential Increase	O(n) (Potential Increase)	O(n)
Performance Stability	Efficient	Efficient	Efficient	Efficient
Scalability	High	High	High (with consideration for large datasets)	High



Agenda

- Introduction
- Data Source
- Environment Setup
- Methodology
  - 1) Data Preprocessing

Results and Discussion

- **Merge Sort Problem**
- **Quick sort Problem**

Problem 2: QuickSort

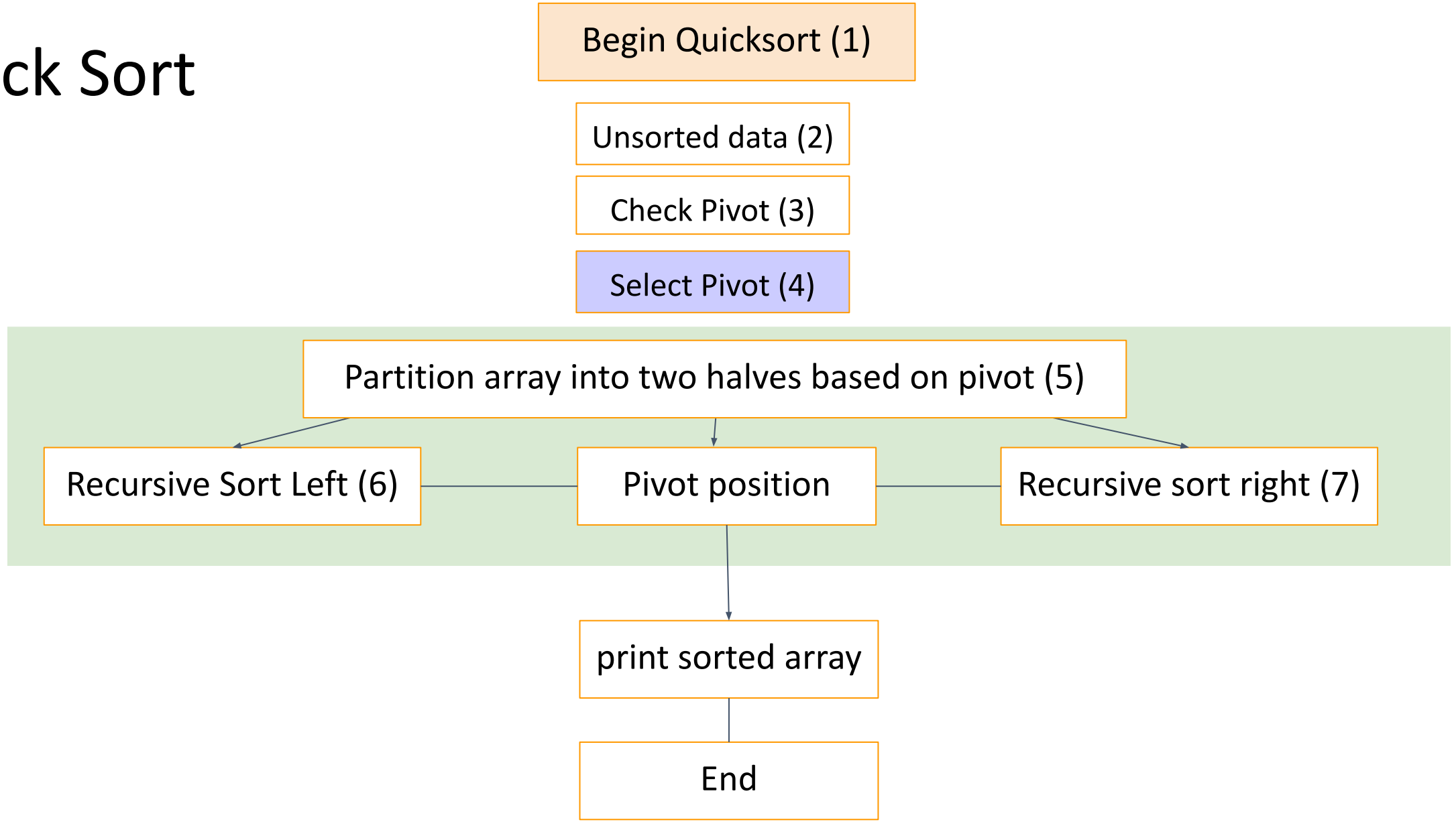
What is Quick sort?

Pseudo Code

- Task 1
- Task 2
- Task 3

Conclusion

Quick Sort


Begin Quicksort (1)
Unsorted data (2)
Check Pivot (3)
Select Pivot (4)
Partition array into two halves based on pivot (5)
Recursive Sort Left (6)
Pivot position
Recursive sort right (7)
print sorted array
End

Merge Sort

What is Quick sort?

Pseudo Code

- Part a
- Part b
- Part c

Conclusion

Pseudocode

```
# 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)
```

Merge Sort

What is Quick sort?

Pseudo Code

- Part a
- Part b
- Part c

Conclusion

Main Code for Quick Sort?

```
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)
```

Quick Sort

What is Quick sort?

Pseudo Code

- Part a
- Part b
- Part c

Conclusion

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

Quicksort Strategies for Sorting City Latitudes

- Divide (Step 1)
- Recursive and Partition (Step 2)
- Combine (Step 3)

Two Quicksort Methods

- Random Pivot Selection
- Fixed Pivot Position

```
# 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]
```

```
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, -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, -43.5833, -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, -40.7333, -40.6219, -40.5833, -40.5725, -40.4, -40.355, -40.3167, -40.2833, -40.2167, -40.1667, -40.1333, -40.1, -40.0667, -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, -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, -38.7433, -38.7333, -38.7167, -38.7
```

```
def partition(array, low, upper):
    comparison_count = 0
    # Selecting last element as pivot.
    pivot = array[upper]
```

```
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, -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, -43.5833, -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, -40.7333, -40.6219, -40.5833, -40.5725, -40.4, -40.355, -40.3167, -40.2833, -40.2167, -40.1667, -40.1333, -40.1, -40.0667, -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, -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,
```

Quick Sort Step	Method 1) Random Pivot Selection	Method 2) Fixed Pivot Position (Last Index)
Divide (Step 1)	Chooses a random pivot from the subarray	Chooses the last element as the pivot
Recursive and Partition (Step 2)	Sorts subarrays; linear time partitioning	Sorts subarrays; linear time partitioning
Combine (Step 3)	Merges sorted subarrays	Merges sorted subarrays
Performance Evaluation		
Pivot Decision	Averts worst-case O(n^2) scenario	Risk of O(n^2) in nearly sorted arrays
Execution Time	Similar to fixed pivot	Similar to random pivot
Memory Usage	Slightly higher (due to randomization)	Slightly lower
Number of Comparisons	Slightly variable	Slightly variable

Quick Sort

What is Quick sort?

Pseudocode

- Part a
- Part b
- Part c

Conclusion

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?

Task Overview

The count number of comparisons of quick sort required for sorting a dataset of 34,657 elements.

I. Count the number of merge

1) Count the number of comparisons: 627,034

II. Does it change if you randomly order the list before sorting?

2) Comparison Count : 678,882

Conclusion : The number of comparisons varies each time, code are executed, due to two main possibility reason :

- Initial data order is crucial for Quicksort's performance
- Randomization increases comparisons due to less optimal pivot placement.

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]
```

Data Condition	Comparison Count	Pivot Efficiency	Performance Impact
Non-randomized	627,034	High (when pivot is near median)	Optimal (fewer comparisons)
Randomized	678,882	Variable (depends on pivot position)	Reduced (more comparisons due to uneven splits)

```
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.
```

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
```

I. Count the number of comparisons and II. Does it change if you randomly order the list before sorting

Quick Sort

What is merge sort?

Pseudocode

- Part a
- Part b
- Part c

Conclusion

Use the latitude and longitude values for each city.

Data : Latitude and Longtitude

Type : List of Tuple

Objective: Sort 44,691 pairs of geographic coordinates (latitude, longitude) using Merge Sort.

Variable name : *'pair_cities_la_and_long'*

Decision making : Sort all pairs, including duplicates, for comprehensive geographic organization.

```
# 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.
```

Quick Sort

What is merge sort?

Pseudocode

- Part a
- Part b
- Part c

Conclusion

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

Approach Overview:

- Sequential sorting by latitude, then longitude.
- Compare longitudes when latitudes are equal.
- Implemented Strategies:

Implemented Strategies:

1) Random Pivot Value:

- Random element as pivot for diversity in partitioning.

2) Last Index Pivot:

- Last element as pivot for consistency in partitioning.

3) Iterative Stack Quicksort:

- Uses a stack and the last element as pivot to avoid recursion.

1) Random Pivot Value

```python
# 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
```

```
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
```

Quicksort with Random Pivot:

- Random pivot selection for latitude-longitude sorting.
- Recursively sorts partitions based on pivot placement.

Function Breakdown:

- Quicksort_lat_lng: Sequentially sorts coordinates.
- Partition_lat_lng: Orders pairs around a random pivot.

Performance Metrics:

- Time and memory usage measured pre- and post-sorting.

Output:

- Displays sorted array with efficiency data.

2) Using 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]

    # 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()
```

Issue Encountered:

- 'RecursionError' after 5 minutes due to maximum recursion depth exceeded.
- Indicates inefficiency with the last element pivot in large datasets.

Cause of Error:

- Inefficient pivot choice leading to worst-case O(n^2) performance.
- Highly unbalanced recursive calls increasing with array size.

Potential Solutions:

- Exploring alternative pivot selections: random, mean, or median.
- Aiming for balanced partition sizes to optimize performance.

Considerations:

- Performance also dependent on external factors like system environment and concurrent processes.

3) Iterative Stack Quicksort with Last Element Pivot.

```python
def quicksort_lat_lng_iterative(cities):
    if not cities:
        return  # Handling empty list
    stack = [(0, len(cities) - 1)]
    while stack:
        low, high = stack.pop()
        if low < high:
            pivot_index = partition_lat_lng_lastIndexPivot(cities, low, high)
            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
count_start_time_random = time.time()
peak_memory_before = max(memory_usage(-1, interval=0.1, timeout=1))
quicksort_lat_lng_iterative(cities)  # Use the iterative version here
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('')
```

```
Execution time: 292.03499603271484 seconds
----------------------------------------
Peak memory usage: 67.078125 MiB
----------------------------------------

Sorted cities: [('Puerto Williams', -54.9333, -67.6167), ('Ushuaia', -54.8019, -68.3031), ('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), ('Veintiocho de Noviembre', -51.65, -72.3), ('Río Gallegos', -51.6233, -69.216
1), ('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), ('P
uerto Deseado', -47.75, -65.9167), ('Cochrane', -47.2547, -72.575), ('Halfmoon Bay', -46.9, 168.1333), ('Pico Truncado', -46.79
5, -67.955), ('Perito Moreno', -46.5886, -70.9242), ('Las Heras', -46.55, -68.95), ('Caleta Olivia', -46.4333, -67.5333), ('Inv
ercargill', -46.4131, 168.3475), ('Glencoe', -46.1833, 168.6833), ('Rada Tilly', -45.9333, -67.5333), ('Mosgiel', -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, 167.7167), ('Puerto Aysén', -45.4, -72.6833), ('A
lto Río Senguer', -45.0167, -70.8167), ('Macetown', -44.865, 168.819), ('Wanaka', -44.7, 169.15), ('Timaru', -44.3931, 171.250
8), ('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, -73.5
961), ('Kingston', -42.9769, 147.3083), ('Esquel', -42.9, -71.3167), ('Hobart', -42.8806, 147.325), ('Puerto Madryn', -42.7667,
```

Background:

- Shifted to an iterative stack-based approach due to 'RecursionError' with the last element pivot.

Implementation:

- Adapted from GeeksforGeeks (2012).
- Successfully sorted the dataset by latitude and longitude without recursion issues.

Performance Metrics:

- Execution time: Approximately 500 seconds.
- Comparisons: Around 1 million.
- Memory usage: Comparable to other methods.

Part C : Summary of Quicksort Implementations

Pivot Strategy	Execution Time	Memory Usage	Error Handling	Performance Notes
Random Pivot	Balanced	Moderate	None	More balanced splits, improved overall performance
Last Element Pivot	Varies	Moderate	Prone to 'RecursionError'	Can lead to unbalanced splits, risk of inefficiency
Iterative Stack with Last Element Pivot	Longer	Comparable	Avoids 'RecursionError'	Longer execution times, addresses recursion depth issues

Overall Summary QuickSort:

Operation	Execution Time (seconds)	Peak Memory Usage (MiB)	Observations
Quick sort with Random Pivot	1.32081	73.58593	More efficient with balanced partitions, demonstrating faster execution comparable to the basic QuickSort.
Quick sort with Last Element Pivot	1.258035	77.6679	Slight time improvement but increased memory usage indicates a trade-off between speed and resource use.
Quick sort with Random Pivot (after random reordering)	1.217647	72.91796	Efficiency remains consistent even after randomizing, showing resilience of the random pivot method.
Quick sort using Pivot Value	1.460224	78.6875	Slightly longer execution time, indicating that pivot value choice can affect sorting speed.
Quick sort using Last Element Pivot	Error	Error	Encountered 'RecursionError', suggesting this method may lead to inefficiency in certain environments.
Quick sort by distance with Stack Method	346.5783	156.9765	Significantly higher execution time and memory usage, showing it may not be optimal for all datasets

Conclusion

Introduction

Data Source

Environment setup

Methodology

1) Data Preprocessing

Results and Discussion

- **Merge Sort Problem**
- **Quick sort Problem**

Overall observation

Conclusion

Sorting Algorithm	Efficiency	Adaptability	Notable Insights
Merge Sort	High (O(n log n) time/space complexity)	Robust (performance consistent regardless of data order)	Effective for sequential latitude and longitude sorting
QuickSort	Varied (dependent on pivot selection)	Sensitive (to initial order, especially with randomization)	Random pivot enhances execution speed and reduces memory usag

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

References

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