Assignment – 1

1. **Trade offs**
   a) **Distributed Computing vs. Centralized System:** A centralized system is one where all important tasks like data processing, storage, and decision-making are handled by a single main computer or server. This central point controls and manages all resources and functionalities of the system. In a distributed system, different parts of the system are located on different computers or devices connected together. Each part can operate independently but works together to achieve common goals**. (Geek of Geeks)**
   b) **Scaling Vertically vs. Horizontally:** Vertical scaling, also known as scaling up, involves increasing the capacity of a single server or virtual machine (AWS EC2 in our case) by adding more resources, such as CPU and RAM. **Horizontal scaling**, or scaling out, involves **adding more machines** to distribute the load across multiple servers It presents a range of advantages that make it an appealing choice for apps seeking a scalable solution. **(**https://medium.com)
   c) **Batch vs. Stream Processing:** Batch processing involves executing data processing jobs on large volumes of data in chunks or batches. This method is particularly effective for resource-intensive jobs, repetitive tasks, and managing extensive datasets where real-time processing isn't required. Stream processing, also known as real-time data processing, handles data in real-time as it arrives, allowing for immediate analysis and action. This method is ideal for applications where timely data processing is critical. **(Geek of Geeks)**
   d) **Monolithic vs. Microservices:** A monolithic architecture is a traditional approach where an entire application is built as a single, indivisible unit. All components, such as the user interface, business logic, and data access layer, are tightly integrated and deployed together. This architecture is characterized by its simplicity and ease of development, especially for small to medium-sized applications. A **microservices architecture** breaks down an application into smaller, independent services, each representing a specific business capability. These services are loosely coupled and communicate over a network, often using lightweight protocols like HTTP. Each service can be developed, deployed, and scaled independently. **(Geek of Geeks)**
   e) **Inter-Service Communication (EDA or REST API):** Inter-service communication in microservices refers to the way different, independent services within a microservices architecture interact and share data to perform a cohesive function. In a microservices architecture, an application is broken down into smaller, self-contained services, each responsible for a specific piece of functionality, like user authentication, order processing, or inventory management. These services must communicate with each other to accomplish broader tasks. This communication can be synchronous, where one service waits for a response from

another (using methods like RESTful APIs or gRPC), or asynchronous, where services send messages without waiting for immediate responses (using message brokers like Kafka or RabbitMQ). The choice of communication method impacts the system's performance, scalability, and fault tolerance, making it a crucial aspect of designing a robust microservices architecture. Properly managing inter-service communication is essential to ensure that the overall system functions smoothly and can handle the complexities of distributed systems.**(Geek of geeks)**

2. **Search in an Unsorted Dataset**
   Since the data is unsorted, using Binary Search is not an option (since it requires sorting first).
   Linear Search: Simple but slow (O(n) time complexity)
   Hashing: If frequent lookups are needed, we can use a hash table to store elements (O(1) lookup, but requires extra space)
   Sorting First vs Direct Search: Sorting takes O(n log n) but allows binary search for future queries.

3. **Return vs Yield:**
   A Python function that generates 100 random numbers and demonstrate both return and yield.

```
import random
# Using return
def generate_numbers_return():
    numbers = [random.randint(1, 1000) for _ in range(100)]
    return numbers  # Returns the entire list at once

# Using yield
def generate_numbers_yield():
    for _ in range(100):
        yield random.randint(1, 1000)  # Generates numbers lazily

# Observations:
numbers_return = generate_numbers_return()
numbers_yield = generate_numbers_yield()

print(type(numbers_return))  # list
print(type(numbers_yield))  # generator
```

4. **Merge Sort with Batch Processing & MapReduce:**
   - Generate the numbers using generate_numbers_return()
   - Sort using MergeSort
   - Implement Batch Processing
   - Attempt MapReduce Paradigm

   Merge Sort Implementation:

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return merge(left, right)

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

# Generate numbers and sort
numbers = generate_numbers_return()
sorted_numbers = merge_sort(numbers)
```

Batch Processing:

```
batch_size = 10
batches = [numbers[i:i+batch_size] for i in range(0, len(numbers), batch_size)]
sorted_batches = [merge_sort(batch) for batch in batches]
```

MapReduce Approach:

```
 from multiprocessing import Pool

def parallel_sort(batch):
    return merge_sort(batch)

if __name__ == "__main__":
    with Pool() as pool:
sorted_batches = pool.map(parallel_sort, batches)
```

5. **Big Data Usecase:**

<u>Steps:</u>

1. Collect INR to USD data for past X years
    - Use an API (like exchangeratesapi.io)

   - Web scraping using BeautifulSoup
   - Manually download CSV and process it
2. Analyze the Trend
   - Find peaks in INR depreciation/appreciation.
   - Compare against EUR/USD for additional insights.
3. Implement Batch Processing
   - Process data in chunks rather than loading all at once.
4. Use MapReduce Paradigm
   - Distribute data analysis tasks across multiple processors.

Example Python Script for Data Collection

```
import requests
import csv
from datetime import datetime, timedelta

API_KEY = "your_api_key_here"
BASE_URL = "https://api.exchangeratesapi.io/history"

def fetch_exchange_rates(start_date, end_date):
    params = {
        "start_at": start_date,
        "end_at": end_date,
        "base": "USD",
        "symbols": "INR"
    }
    response = requests.get(BASE_URL, params=params)
    data = response.json()

    with open("inr_usd.csv", "w", newline="") as file:
        writer = csv.writer(file)
        writer.writerow(["Date", "ExchangeRate"])
        for date, rates in data["rates"].items():
            writer.writerow([date, rates["INR"]])

fetch_exchange_rates("2020-01-01", "2023-01-01")
```

Finding Peaks in INR Depreciation

```
import pandas as pd
import numpy as np

data = pd.read_csv("inr_usd.csv")
data["Date"] = pd.to_datetime(data["Date"])
data = data.sort_values("Date")

# Finding peaks (sudden INR weakening)
data["Diff"] = data["ExchangeRate"].diff()
peaks = data[data["Diff"] > data["Diff"].quantile(0.95)]  # Top 5% changes
```

```
print(peaks)
```

Batch Processing

```
chunk_size = 50
for chunk in pd.read_csv("inr_usd.csv", chunksize=chunk_size):
  print(chunk.head())
```

MapReduce for Currency Comparison

```
from multiprocessing import Pool

def process_currency_chunk(chunk):
   chunk["Diff"] = chunk["ExchangeRate"].diff()
   return chunk[chunk["Diff"] > chunk["Diff"].quantile(0.95)]  # Filter
peaks

chunks = pd.read_csv("inr_usd.csv", chunksize=50)

with Pool() as pool:
     results = pool.map(process_currency_chunk, chunks)
```