Assignment - Big data

1. Trade-offs

a. Time Complexity vs. Space Complexity

- Description: In algorithmic design, optimizing for faster execution time often requires additional memory usage, while optimizing for memory efficiency may result in slower performance.
- Perspectives:
    - Developer Perspective: Prioritizing time complexity is crucial for real-time systems (e.g., gaming, financial trading), where speed is paramount.
    - System Perspective: In memory-constrained environments (e.g., embedded systems), space complexity takes precedence.
- Example: Dynamic programming often trades space for time by storing intermediate results to avoid redundant computations.

b. Consistency vs. Availability in Distributed Systems

- Description: The CAP theorem states that in the presence of network partitions, a distributed system cannot simultaneously guarantee consistency and availability.
- Perspectives:
    - Consistency-First Systems: Databases like PostgreSQL prioritize consistency, ensuring all nodes see the same data at the same time.
    - Availability-First Systems: Systems like Apache Cassandra prioritize availability, allowing reads and writes even during network partitions .
- Example: Financial systems often favor consistency to prevent discrepancies, while social media platforms prioritize availability for uninterrupted user experience.

c. Performance vs. Maintainability in Software Engineering

- Description: High-performance code often involves optimizations that increase complexity, making it harder to maintain and extend.
- Perspectives:
    - Short-Term Projects: Performance may take precedence in time-sensitive projects (e.g., prototypes).
    - Long-Term Projects: Maintainability is critical for scalable and sustainable software.

- Example: Using low-level languages like C++ for performance vs. high-level languages like Python for maintainability.

d. Security vs. User Experience

- Description: Enhanced security measures (e.g., multi-factor authentication) can introduce friction, reducing usability.
- Perspectives:
  - High-Security Systems: Banking applications prioritize security, even at the cost of user convenience.
  - User-Centric Systems: Social media platforms balance security with seamless user experience.
- Example: Passwordless login systems (e.g., biometrics) aim to reduce friction while maintaining security.

e. Cost vs. Quality in Software Development

- Description: Higher quality software typically requires more resources, increasing costs.
- Perspectives:
  - Budget-Constrained Projects: Startups may prioritize cost over quality to deliver quickly.
  - Mission-Critical Systems: Aerospace or healthcare systems prioritize quality, regardless of cost.

Example: Open-source tools may reduce costs but require additional effort to ensure quality.

2. Search

a. Linear Search

- Description: Scans each element sequentially until a match is found or the dataset is exhausted.
- Advantages: Simple to implement, no preprocessing required.
- Disadvantages: O(n) complexity makes it inefficient for large datasets.
- Use Case: Suitable for small, unsorted datasets.

b. Sorting First + Binary Search

- Description: Sort the dataset first (O(n log n)), then perform binary search (O(log n)).
- Advantages: Efficient for repeated searches on the same dataset.

- Disadvantages: High upfront cost for sorting.
- Use Case: Ideal for static datasets with frequent queries.

c. Hashing

- Description: Use hash functions to achieve O(1) average-case lookup time.
- Advantages: Extremely fast for lookups.
- Disadvantages: Requires additional memory; collisions can degrade performance.
- Use Case: Best for scenarios where memory overhead is acceptable.

d. Tree-Based Search Structures

- Description: Balanced trees (e.g., AVL, B-Trees) provide O(log n) search efficiency.
- Advantages: Efficient for dynamic datasets with frequent insertions/deletions.
- Disadvantages: Slightly higher complexity than hashing.
- Use Case: Ideal for database indexing and dynamic datasets.

Comparative Analysis

- Small, Unsorted Dataset: Linear Search suffices due to simplicity.
- Frequent Queries on Static Dataset: Sorting + Binary Search or Tree-Based Indexing.
- Memory-Intensive Applications: Hashing for O(1) lookups.
- Dynamic Datasets: Tree-Based Approaches for balanced performance.

3. Return vs. Yield

```
import random


def generate_numbers_return():
    return [random.randint(1, 1000) for _ in range(100)]


def generate_numbers_yield():
    for _ in range(100):
        yield random.randint(1, 1000)
```

Observations

- Return:
  - Compiles and returns the entire dataset at once.
  - Consumes more memory as the entire list is stored in memory.
  - Suitable for small datasets or when all data is needed immediately.
- Yield:
  - Supports lazy evaluation, generating values on-the-fly.
  - Conserves memory by producing one value at a time.
  - Ideal for large datasets or streaming applications.

4. MergeSort

a. Generate Data:

```
data = generate_numbers_return()
```

b. Implement Merge Sort

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

        merge_sort(left)
        merge_sort(right)

        i = j = k = 0
        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                arr[k] = left[i]
                i += 1
            else:
                arr[k] = right[j]
                j += 1
```

```
            k += 1

        while i < len(left):
            arr[k] = left[i]
            i += 1
            k += 1
        while j < len(right):
            arr[k] = right[j]
            j += 1
            k += 1
    return arr

# Sort the dataset
sorted_data = merge_sort(data)
print(sorted_data)
```

c. Integrate Batch Processing

- Description: Divide the dataset into smaller batches, sort each batch individually, and then merge the results.
- Advantages: Reduces memory overhead and improves scalability for large datasets.

d. Explore the MapReduce Paradigm

- Description: Use a distributed framework (e.g., Hadoop) to parallelize sorting across multiple nodes.
- Steps:
  1. Map Phase: Divide the dataset into chunks and sort each chunk locally.
  2. Reduce Phase: Merge the sorted chunks into a final sorted dataset.
- Advantages: Highly scalable for massive datasets.

6. git hub link:

https://github.com/udaykiran017/Assignment_Bigdata