DS2030 Data Structures and Algorithms for Data Science
Week-10 Practice Set

October 28th, 2025

1 Merge Sort

Problem Statement

Merge Sort is a classic divide-and-conquer sorting algorithm that divides the input array into two halves, recursively sorts them, and then merges the two sorted halves. This approach ensures a stable time complexity of O(n log n) in all cases. The merging step is crucial, as it combines the subarrays while maintaining the sorted order. Given an array of integers, implement Merge Sort to sort it efficiently.

Tasks

Implement the Merge Sort algorithm with the following functionalities:

1. **Divide and Conquer Visualization** Function: `merge_sort(arr)` Where `arr` is a list of integers. Recursively divide the array and return the sorted array. Print the recursion tree or steps showing how the array is divided into subarrays and merged back.
2. **Merge Step Implementation** Function: `merge(left, right)` Implement the merge function that combines two sorted subarrays into one sorted array. Print the merge process, showing comparisons and placements.
3. **Efficiency Observation** Print the sorted array and the time/space complexity of the algorithm.

Input

```
arr = [38, 27, 43, 3, 9, 82, 10]
```

Expected Output

```
Merge Sort Steps:
Divide: [38, 27, 43, 3, 9, 82, 10] -> [38, 27, 43] and [3, 9, 82, 10]
Divide: [38, 27, 43] -> [38] and [27, 43]
Divide: [27, 43] -> [27] and [43]
Merge: [27] and [43] -> [27, 43] (comparisons: 27<43)
Merge: [38] and [27, 43] -> [27, 38, 43] (comparisons: 38>27, 38<43)
Divide: [3, 9, 82, 10] -> [3, 9] and [82, 10]
Divide: [3, 9] -> [3] and [9]
Merge: [3] and [9] -> [3, 9] (comparisons: 3<9)
Divide: [82, 10] -> [82] and [10]
Merge: [82] and [10] -> [10, 82] (comparisons: 82>10)
Merge: [3, 9] and [10, 82] -> [3, 9, 10, 82] (comparisons: 3<10, 9<10, 9<82)
Merge: [27, 38, 43] and [3, 9, 10, 82] -> [3, 9, 10, 27, 38, 43, 82] (comparisons:
    27>3, 27>9, 27>10, 38>27, 43>38, 82>43)
Sorted array: [3, 9, 10, 27, 38, 43, 82]
Time complexity: O(n log n), Space: O(n)
```

2 Quick Sort

Problem Statement

Quick Sort is a divide-and-conquer sorting algorithm that selects a 'pivot' element from the array and partitions the other elements into two sub-arrays, according to whether they are less than or greater

than the pivot. The sub-arrays are then recursively sorted. Quick Sort is efficient with an average time complexity of O(n log n), though it can degrade to $O(n^2)$ in the worst case with poor pivot selection. Given an array of integers, implement Quick Sort to sort it efficiently.

Tasks

Implement the Quick Sort algorithm with the following functionalities:

1. **Partition Visualization** Function: `quick_sort(arr)` Where `arr` is a list of integers. Recursively partition the array using the last element as pivot and return the sorted array. Print the recursion steps showing how the array is partitioned into subarrays.
2. **Partition Step Implementation** Function: `partition(arr, low, high)` Implement the partition function that rearranges the array around the pivot. Print the partition process, showing swaps and pivot placement.
3. **Efficiency Observation** Print the sorted array and the time/space complexity of the algorithm.

Input

```
arr = [38, 27, 43, 3, 9, 82, 10]
```

Expected Output

```
Quick Sort Steps:
Partition: [38, 27, 43, 3, 9, 82, 10] (pivot: 10) -> [3, 9] [38, 27, 82, 43] pivot at
    index 2 (comparisons: 38>10, 27>10, 43>10, 3<10 swap, 9<10 swap, 82>10)
Partition: [3, 9] (pivot: 9) -> [3] [] pivot at index 1 (comparisons: 3<9)
Partition: [38, 27, 82, 43] (pivot: 43) -> [38, 27] [82] pivot at index 2 (comparisons
    : 38<43, 27<43, 82>43)
Partition: [38, 27] (pivot: 27) -> [] [38] pivot at index 0 (comparisons: 38>27)
Sorted array: [3, 9, 10, 27, 38, 43, 82]
Time complexity: O(n log n) average, O(n^2) worst; Space: O(log n)
```

3 Insertion Sort

Problem Statement

Insertion Sort is a simple comparison-based sorting algorithm that builds the final sorted array one item at a time. It is much like sorting playing cards in your hands, where each new card is inserted into the correct position among the previously sorted cards. It has a time complexity of $O(n^2)$ but performs well on small or nearly sorted arrays. Given an array of integers, implement Insertion Sort to sort it efficiently for small datasets.

Tasks

Implement the Insertion Sort algorithm with the following functionalities:

1. **Insertion Visualization** Function: `insertion_sort(arr)` Where `arr` is a list of integers. Iteratively insert each element into its correct position in the sorted subarray. Print the steps showing how each element is inserted with comparisons and shifts.
2. **Insertion Step Implementation** Function: `insert(arr, i)` Implement the insertion for the i-th element, shifting larger elements right. Print the shifts and comparisons for each insertion.
3. **Efficiency Observation** Print the sorted array and the time/space complexity of the algorithm.

Input

```
arr = [38, 27, 43, 3, 9, 82, 10]
```

Expected Output

```
Insertion Sort Steps:
Insert 38: [38]
Insert 27: compare 27<38, shift 38 -> [27, 38]
Insert 43: compare 43>38, no shift -> [27, 38, 43]
Insert 3: compare 3<43 shift, 3<38 shift, 3<27 shift -> [3, 27, 38, 43]
Insert 9: compare 9<43 shift, 9<38 shift, 9<27 shift, 9>3 no shift -> [3, 9, 27, 38,
    43]
Insert 82: compare 82>43, no shift -> [3, 9, 27, 38, 43, 82]
Insert 10: compare 10<82 shift, 10<43 shift, 10<38 shift, 10<27 shift, 10>9 no shift
    -> [3, 9, 10, 27, 38, 43, 82]
Sorted array: [3, 9, 10, 27, 38, 43, 82]
Time complexity: O(n^2), Space: O(1)
```

4 Breadth-First Search (BFS)

Problem Statement

Breadth-First Search (BFS) is a graph traversal algorithm that explores all vertices level by level, starting from a source vertex. It uses a queue to visit neighbors before moving to the next level, making it ideal for finding the shortest path in an unweighted graph. This algorithm computes the shortest path in terms of the number of edges. Given an undirected graph represented as an adjacency list and a starting vertex, perform BFS to find the shortest paths to all other vertices.

Tasks

Implement the BFS algorithm with the following functionalities:

1. **BFS Traversal and Levels** Function: `bfs(graph, start)` Where `graph` is a dictionary representing the adjacency list and `start` is the starting vertex. Compute and return distances (levels) from the start vertex to all others. Print the BFS traversal order and levels (L0, L1, etc.).
2. **Shortest Path Reconstruction** Function: `shortest_path(graph, start, end)` Using BFS, reconstruct and print the shortest path from `start` to `end`.
3. **Efficiency Observation** Print the distances dictionary and the time/space complexity.

Input

```
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}
start = 'A'
```

Expected Output

```
BFS Traversal from A:
L0: A
L1: B, C
L2: D, E, F
Distances: {'A': 0, 'B': 1, 'C': 1, 'D': 2, 'E': 2, 'F': 2}
Shortest path from A to F: A -> C -> F
Time complexity: O(V + E), Space: O(V)
```