Lab 01: Introduction to Algorithm Analysis and Design

ICT 2202 Algorithm Analysis and Design Lab

Submitted By

Name
Md. Shaon Khan

Class Roll
1984

Submitted To

Professor Fahima Tabassum, PhD

Institute of Information Technology

Jahangirnagar University

Submission Date: 14/10/2025

Merge Sort

1.1 **Overview:**

Merge Sort is a divide-and-conquer sorting algorithm that works by recursively dividing an array into smaller subarrays until each subarray contains a single element, then merging those subarrays back together in sorted order. The key operation is the merge function, which combines two sorted arrays into a single sorted array. Merge Sort has a time complexity of O(n log n) in all cases and requires O(n) additional space.

Algorithm:

1. Algorithm for mergeSort(arr, left, right)

This function is responsible for recursively splitting the array into smaller halves until they are tri-vially sorted (size of 1), and then calling the merge function to combine them.
Input:

- arr: The array to be sorted.
- left: The starting index of the current sub-array.
- right: The ending index of the current sub-array.

Steps:

1. Check Base Case: If the left index is less than the right index, it means the current sub-array has more than one element. Proceed. Otherwise, the single-element sub-array is al-ready sorted, so simply return.
2. Find Mid-Point: Calculate the middle index mid of the current sub-array.
    - mid = (left + right) // 2
3. Recursively Sort First Half: Call mergeSort(arr, left, mid) to sort the left sub-array from left to mid.
4. Recursively Sort Second Half: Call mergeSort(arr, mid + 1, right) to sort the right sub-array from mid + 1 to right.
5. Merge the Sorted Halves: Call merge(arr, left, mid, right) to combine the two now-sorted halves (left...mid and mid+1...right) back into the original array in sorted order.

2.Algorithm for merge(arr, left, mid, right)

This function merges two adjacent, sorted sub-arrays into a single sorted sub-array.

Input:

- arr: The original array containing the two sub-arrays to be merged.
- left: The starting index of the first sub-array.
- mid: The ending index of the first sub-array. (The second sub-array starts at mid + 1).
- right: The ending index of the second sub-array.

Steps:

1. Calculate Sizes:
    - Let n1 = mid - left + 1 (size of the left sub-array).
    - Let n2 = right - mid (size of the right sub-array).
2. Create Temporary Arrays:
        - Create two temporary arrays, L of size n1 and R of size n2.
    - Copy Data:
        - Copy the elements from arr[left ... mid] into the temporary array L.
        - Copy the elements from arr[mid + 1 ... right] into the temporary array R.
3. Merge Back into Original Array:
    - Initialize three pointers:
        - i = 0 (pointer for L)
        - j = 0 (pointer for R)
        - k = left (pointer for the main arr where the next sorted element will be placed)
    - While both i < n1 and j < n2:
        - Compare the current element of L[i] with the current element of R[j].
        - If L[i] <= R[j]:
            - Copy L[i] to arr[k].
            - Increment i and k.

        - Else:
            - Copy R[j] to arr[k].
            - Increment j and k.

4. Copy Remaining Elements:
    - If any elements remain in the left array L (i.e., i < n1), copy all remaining elements from L to arr.
    - If any elements remain in the right array R (i.e., j < n2), copy all remaining elements from R to arr.

1.2 Problem 01:

Implement Merge Sort to sort an array of integers in non-decreasing order.

1.2.1 Code

```
def merge(arr, left, mid, right):
    n1 = mid - left + 1
    n2 = right - mid

    L = [0] * n1
    R = [0] * n2

    for i in range(n1):
        L[i] = arr[left + i]
    for j in range(n2):
        R[j] = arr[mid + 1 + j]

    i = 0
    j = 0
    k = left

    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1

    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

def mergeSort(arr, left, right):
    if left < right:
        mid = (left + right) // 2

        mergeSort(arr, left, mid)
        mergeSort(arr, mid + 1, right)
        merge(arr, left, mid, right)
```

```
if __name__ == "__main__":
    arr = [12, 11, 13, 5, 6, 7]
    print("Original array:", arr)

    mergeSort(arr, 0, len(arr) - 1)
    print("Sorted array:", arr)
```

1.2.2 Output:

Original array: [12, 11, 13, 5, 6, 7]
Sorted array: [5, 6, 7, 11, 12, 13]

1.2.3 Explanation:

The Merge Sort algorithm works as follows:

1. Divide Phase: The array [12, 11, 13, 5, 6, 7] is recursively divided:
    - First division: [12, 11, 13] and [5, 6, 7]
    - Second division: [12, 11], [13], [5, 6], [7]
    - Third division: [12], [11], [13], [5], [6], [7]
2. Conquer/Merge Phase: The single-element arrays are merged back in sorted order:
    - Merge [12] and [11] → [11, 12]
    - Merge [11, 12] and [13] → [11, 12, 13]
    - Merge [5] and [6] → [5, 6]
    - Merge [5, 6] and [7] → [5, 6, 7]
    - Final merge: [11, 12, 13] and [5, 6, 7] → [5, 6, 7, 11, 12, 13]

Discussion:

1. Time Complexity

Best Case: O(n log n)

- Scenario: The array is already completely sorted.
- Analysis: Even if the array is sorted, the algorithm will still perform all the divisions until it reaches single elements. The merge process will still need to compare elements from both halves, though it might do so efficiently. The fundamental divide-and-conquer structure remains unchanged, leading to O(n log n) operations.

Worst Case: O(n log n)

- Scenario: The array is in reverse order, or any arrangement that forces maximum comparisons during the merge step.
- Analysis: Merge Sort's dividing pattern is independent of the input order. Every element will be part of log n split-and-merge levels, and at each level, a total of n comparisons and copies are made across all merges. This consistent behavior guarantees O(nlogn) performance.

Average Case: O(n log n)

- Scenario: The array is in a random order.
- Analysis: Regardless of the initial order of elements, the algorithm will always follow the same division pattern. The number of levels remains logarithmic, and the work per level remains linear. Therefore, the average case is also O(n log n).

2. Space Complexity

O(n)

- Primary Reason: The merge function creates temporary arrays L and R to hold the two sub-arrays being merged. The maximum size of these temporary arrays combined is n (this happens at the very first merge call after dividing single elements).
- Auxiliary Space: This is the key point. Merge Sort is not an in-place sorting algorithm. It requires significant additional memory proportional to the input size.
- Call Stack: The recursive calls use O(log n) space on the call stack. However, this is dominated by the O(n) space required for the temporary arrays, so the overall space com-plexity is O(n).

3.Stability

Merge Sort is stable (equal elements maintain their relative order)

4.Not In-Place

Requires additional memory proportional to the input size

The algorithm is particularly useful for sorting linked lists and external sorting (when data doesn't fit in memory). While it has better time complexity than simpler algorithms like Bubble Sort or Insertion Sort, the space overhead can be a disadvantage for large datasets in memory-constrained environments.