Faculty of Information Technology
University of Moratuwa

BSc(Hons) in Information Technology
BSc(Hons) in Artificial Intelligence

IN 1111 - Data Structures and Algorithm part 2

Level 2 - Semester 2 | Lecture Note 01

Merge Sort

1.1 What is Merge Sort ?

- Merge Sort is a popular sorting algorithm that follows the **divide-and-conquer paradigm.**
- It works by dividing the unsorted list into **'n' sub-lists**, each containing one element, and then repeatedly **merging sub-lists** to produce new sorted sub-lists until there is only one sub-list remaining, which is the sorted list.
- Here's a high-level description of the Merge Sort algorithm:
  - **Divide:** Divide the unsorted list into 'n' sub-lists, each containing one element.
  - **Conquer:** Repeatedly merge sub-lists to produce new sorted sub-lists until there is only one sub-list remaining.
  - **Merge:** Merge two sub-lists into a single sorted sub-list. This process is repeated until there is only one sub-list remaining, which is the sorted list.
- The key operation in Merge Sort is the **merging of two sorted sub-lists**. The merge operation compares elements from two sub-lists and places them in sorted order.

> *While Merge Sort has these advantages, it's important to consider other sorting algorithms like Quicksort, Heapsort, or Insertion sort, depending on the specific requirements and characteristics of the data. For example, quicksort may be faster than mergesort for small datasets or datasets with a high degree of pre-sortedness.*

1.2 Why Merge Sort ?

- Merge Sort is considered advantageous over other sorting algorithms in certain scenarios due to its characteristics and performance. Here are some reasons why Merge Sort is preferred in certain situations:
  - **Stability:** Merge Sort is a stable sorting algorithm, meaning that it maintains the relative order of equal elements in the sorted output. This property is crucial in situations where preserving the initial order of equal elements is important.
  - **Predictable Performance:** Merge Sort has a consistent, predictable performance with a time complexity of O(n log n) in the worst, average, and best cases. This makes it suitable for scenarios where a reliable and efficient sorting algorithm is required across various input data distributions.
  - **External Sorting:** Merge Sort is particularly well-suited for external sorting scenarios where data is too large to fit into memory. It can efficiently handle large datasets by reading and writing smaller chunks of data at a time.
  - **Parallelization:** Merge Sort can be easily parallelized, taking advantage of multiple processors or cores to enhance performance. Each sub-list can be sorted independently, and the sorted sub-lists can be efficiently merged in parallel.
  - **Consistent Memory Usage:** Merge Sort typically uses a consistent amount of additional memory regardless of the input data size. This makes it more memory-efficient than some other algorithms, such as quicksort, in situations where memory usage is a concern.
  - **Ease of Implementation:** Merge Sort has a straightforward and modular implementation, making it easier to understand and maintain. The divide-and-conquer approach lends itself well to recursive implementations.

1.3 Decomposition

- The MergeSort repeatedly divides the array into **two halves** until we reach a stage where we try to perform MergeSort on a subarray of size 1
- This can be achieved by applying recursion until it stops.

1.4 Merging

- The merge step is the solution to the simple problem of merging two sorted lists(array) to build one large sorted list(array).

- The algorithm maintains three pointers, one for each of the two arrays and one for maintaining the current index of the final sorted array.

```
Have we reached the end of any of the arrays?
    No:
        Compare current elements of both arrays
        Copy smaller element into sorted array
        Move pointer of element containing smaller element
    Yes:
        Copy all remaining elements of non-empty array
```

subarray - 1	subarray - 2	sorted combined array
		
		
		
		
		

Since there are no more elements remaining in the second array, and we know that both the arrays were sorted when we started, we can copy the remaining elements from the first array

1.5 Code

- Our task is to merge two subarrays **A[p..q]** and **A[q+1..r]** to create a sorted array **A[p..r]**. So the inputs to the function are A, p, q and r.
- The merge function works as follows:
  - Create copies of the subarrays **L ← A[p..q]** and **M ← A[q+1..r]**
  - Create three pointers **i, j and k**
    - i. i maintains current index of L, starting at 1
    - ii. j maintains current index of M, starting at 1
    - iii. k maintains the current index of A[p..q], starting at p.
  - Until we reach the end of either L or M, pick the larger among the elements from L and M and place them in the correct position at A[p..q]
  - When we run out of elements in either L or M, pick up the remaining elements and put in A[p..q]

- When we run out of elements in either L or M, pick up the remaining elements and put in A[p..r]

subarray - 1	combined array
	
	

-

```
class MergeSort {

  // Merge two subarrays L and M into arr
  void merge(int[] arr[], int p, int q, int r) {

    // Create L ← A[p..q] and M ← A[q+1..r]
    int n1 = q - p + 1;
    int n2 = r - q;

    int L[] = new int[n1];
    int M[] = new int[n2];

    for (int i = 0; i < n1; i++)
      L[i] = arr[p + i];
    for (int j = 0; j < n2; j++)
      M[j] = arr[q + 1 + j];

    // Maintain current index of sub-arrays and main array
    int i, j, k;
    i = 0;
    j = 0;
    k = p;

    // Until we reach either end of either L or M, pick larger among
    // elements L and M and place them in the correct position at A[p..r]
    while (i < n1 && j < n2) {
      if (L[i] <= M[j]) {
        arr[k] = L[i];
        i++;
      } else {
        arr[k] = M[j];
        j++;
      }
      k++;
    }

    // When we run out of elements in either L or M,
    // pick up the remaining elements and put in A[p..r]
    while (i < n1) {
      arr[k] = L[i];
      i++;
      k++;
    }

    while (j < n2) {
      arr[k] = M[j];
      j++;
      k++;
    }
  }

  // Divide the array into two subarrays, sort them and merge them
  void mergeSort(int arr[], int l, int r) {
    if (l < r) {

      // m is the point where the array is divided into two subarrays
      int m = (l + r) / 2;

      mergeSort(arr, l, m);
      mergeSort(arr, m + 1, r);

      // Merge the sorted subarrays
      merge(arr, l, m, r);
    }
  }

  // Print the array
  static void printArray(int arr[]) {
    int n = arr.length;
    for (int i = 0; i < n; ++i)
      System.out.print(arr[i] + " ");
    System.out.println();
  }

  // Driver program
  public static void main(String args[]) {
    int arr[] = { 6, 5, 12, 10, 9, 1 };

    MergeSort ob = new MergeSort();
    ob.mergeSort(arr, 0, arr.length - 1);

    System.out.println("Sorted array:");
    printArray(arr);
  }
}
```

1.6 Time Complexity

Time Complexity	
Best	O(n*log n)
Worst	O(n*log n)
Average	O(n*log n)
Space Complexity	O(n)
Stability	Yes

1.6 Additional Notes