# Fundamentals of Algorithm Design and Machine Learning

Lecture scribe notes

Date: 14th Jan, 2025

Topics Covered

- Basics of Divide and Conquer
- Sorting
    1. Selection Sort
    2. Insertion Sort
    3. Merge Sort
    4. Quick Sort
- Merging K sorted lists

Submitted by: Anupam Das

24BM6JP09

## 1. Introduction to Sorting Algorithms

- What is Sorting?

  Sorting is the process of arranging elements in specific order, either ascending or descending. Sorted data is easier to view, perform analyses and feed further into other algorithms (e.g. binary search). There are many different flavours of sorting algorithms such as *bubble, insertion, merge* each with their own nuances and nitty-gritties. Most of them use Divide and Conquer approach to perform sorting operation.

## 2. Divide and Conquer Paradigm

- What is Divide and Conquer?

  Divide and Conquer is a problem-solving technique in computer science and mathematics that involves breaking a complex problem into smaller, more manageable subproblems. These subproblems are solved independently, often recursively, and their solutions are then combined to form the solution to the original problem.

  Divide and Conquer Steps:

  We first think of a recursive formulation and try to figure out how to divide the problem into smaller identical sub-problems and solve the smaller problem and combine to get the answer to the original problem. Broadly there are 3 steps involved.

  1. **Divide:** Keep decomposing the problem into smaller sub-problems until Base Case B(z) is reached.

     $<X_1, X_2, \ldots, X_k>$ D(x) Decompostion step

  2. **Conquer:** Call subroutine on each of these sub-problems and solve from the base case up.

     $Y_1 - f(x_1), Y_2 - f(x_2), \ldots, Y_k - f(x_k)$

  3. **Combine (Recomposition):** Combine the solutions to each of the sub-problems

     $Y - R (Y_1, Y_2, Y_2, \ldots, Y_k)$

Complexity: $\boldsymbol{T(n)} = \sum_{i=1}^{k} \boldsymbol{T(n_i)} + \boldsymbol{D(k)} + \boldsymbol{R(k)}$

Here $T(n_i)$ represents the smaller sub problems, D(k) is the decomposition step, R(k) is the recomposition step.

A good algorithm *will have a balance of T(n), D(k) and R(k).* After the initial naïve solution is formulated, the algorithm must be further evaluated to find the best split and recombination choice to optimize the performance.

## Sorting Algorithms

### a. Selection Sort

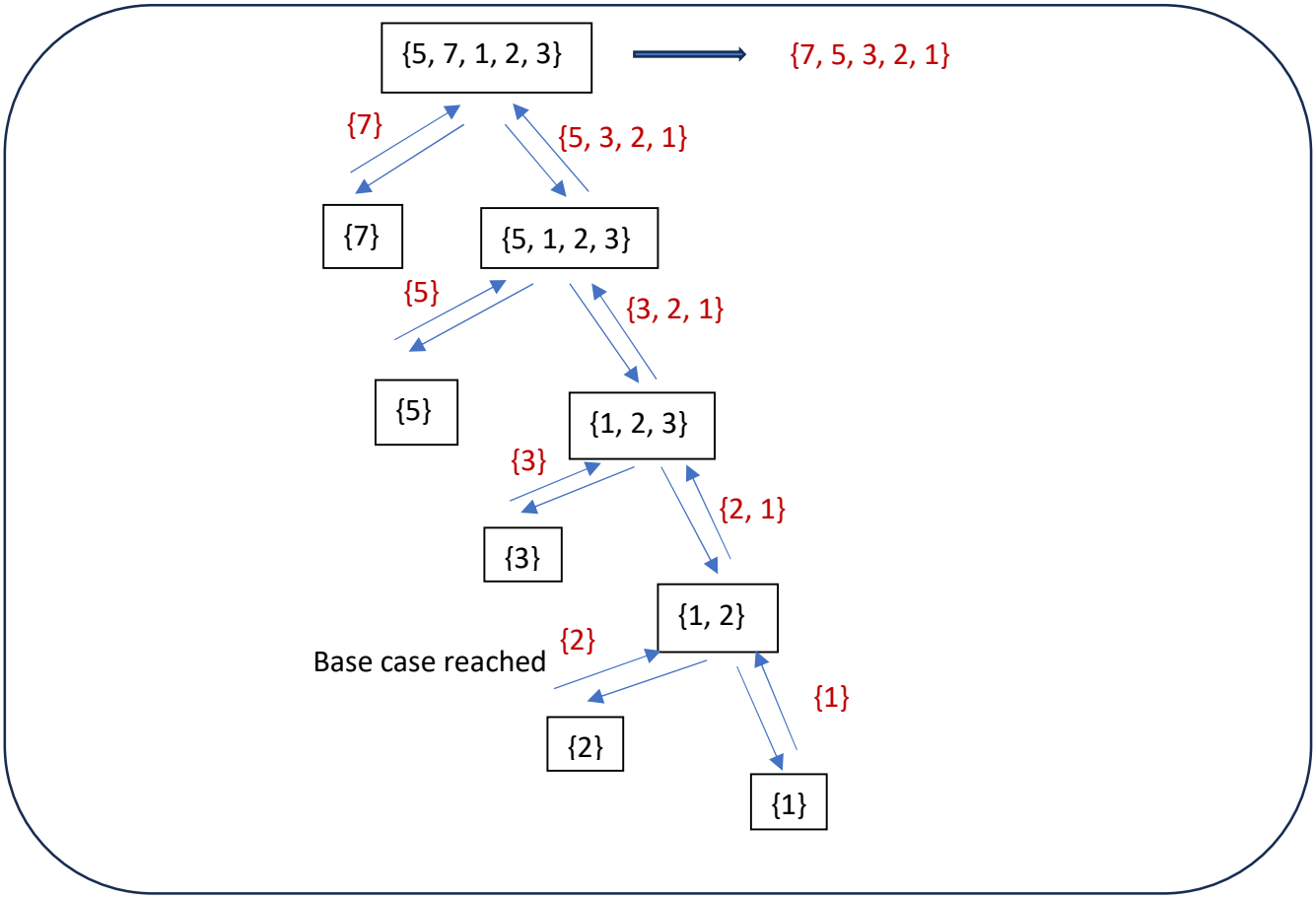
- **Concept and Working Principle**

  Selection Sort is a simple comparison-based sorting algorithm that works by repeatedly selecting the smallest (or largest, depending on order) element from the unsorted part of the list and placing it at the beginning (or end) of the sorted part. It continues until the entire list is sorted. In short remove the max element from the list and compute the rest recursively

- **Pseudocode**

```
Let L = {x1, x2, x3, …,xn)
SelectionSort (L) {
        if (|L| = 1)                    B (Base Case)
                return L
        xi <- findMax (L)               D (Decomposition)
        L' <- L – {xi}
        M <- SelectionSort (L')         C (Calling function recursively)
        return (xi||M)                  R (Recombination)
}
```

- **Recursion Tree Structure**



- **Time Complexity Analysis**

  *T(n) = T(n-1) + O(n)*

  The initial array is broken down into two, one array containing just the max element finding which is O(n) and rest n-1 elements. Solving this we get the time complexity **$O(n^2)$**.

  Here, *findMax* runs over and over again at each step of decomposition, thus worsening the time complexity. Decomposition and recomposition can be played around with e.g. by using a binary tree structure to improve the time complexity.

## b. Insertion Sort

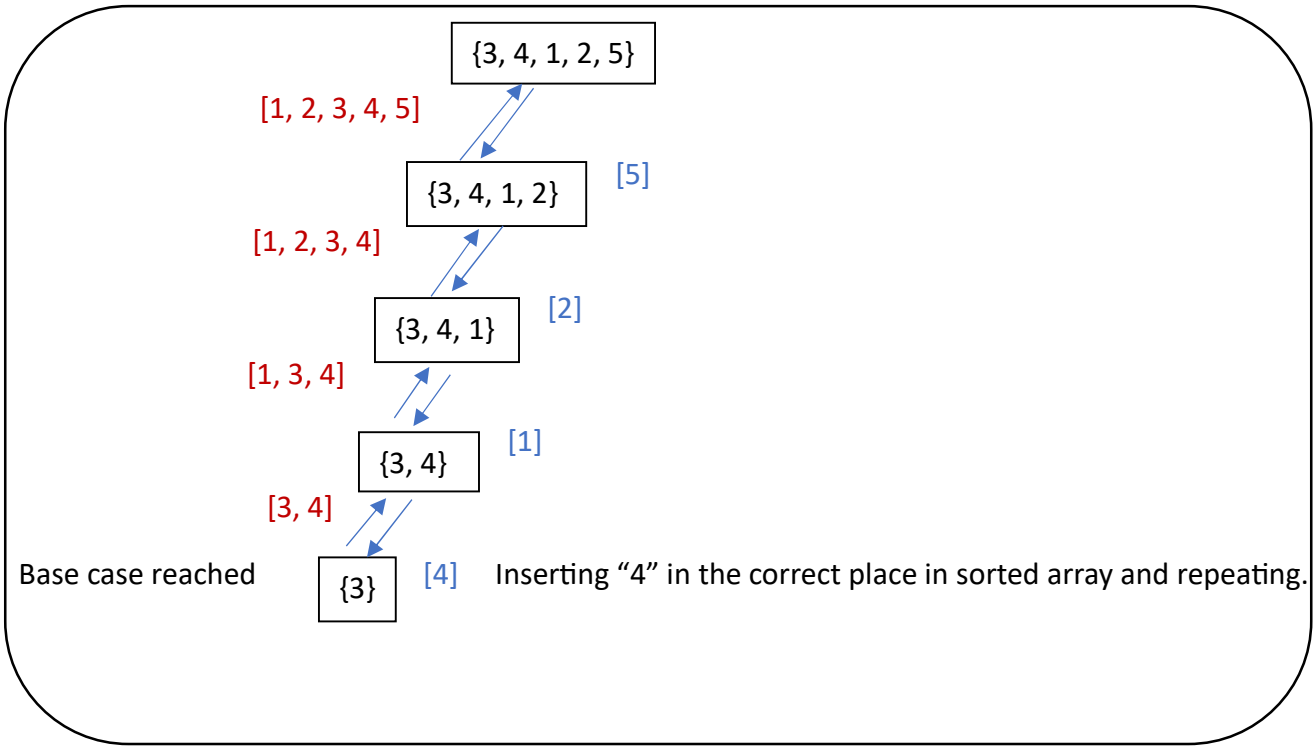
- **Concept and Working Principle**

  Insertion Sort is a simple sorting algorithm that *works by building a sorted portion of the list one element at a time*. It picks an element from the unsorted part and places it in the correct position within the sorted part.

- **Pseudocode**

```
insertionSort (L) {
        if (|L| <= 1)
                return L                    B (Base case)
        M <- insertionSort (L – {x_n})      D (Decomposition)
        M' <- insert (M, x_n)               R (Recombination)
        return M'
}
```

- **Recursion Tree Structure**



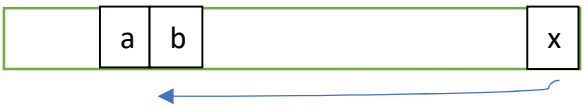
- **Time Complexity Analysis**

  *T(n) = T(n-1) + "Insert"*

  Solving this we get the time complexity as **$O(n^2)$.** We can use a binary tree (AVL) and keep balancing it to optimize this algorithm. *O(nlogn) time to create binary tree and O(1) time for inorder traversal*.

- **Iterative Solution**

  Insertion sort can also be implemented using array and iterative approach.




Find the correct place to insert the last element, which is O(n). Therefore, overall time complexity comes out as $O(n^2)$

## c. Merge Sort

- **Concept and Working Principle**

  Merge Sort is a **divide and conquer** algorithm that recursively breaks down a problem into smaller subproblems until they become simple enough to be solved directly. It is a sorting algorithm that divides the input array into two halves, sorts each half, and then *merges the sorted halves to produce the sorted array.*

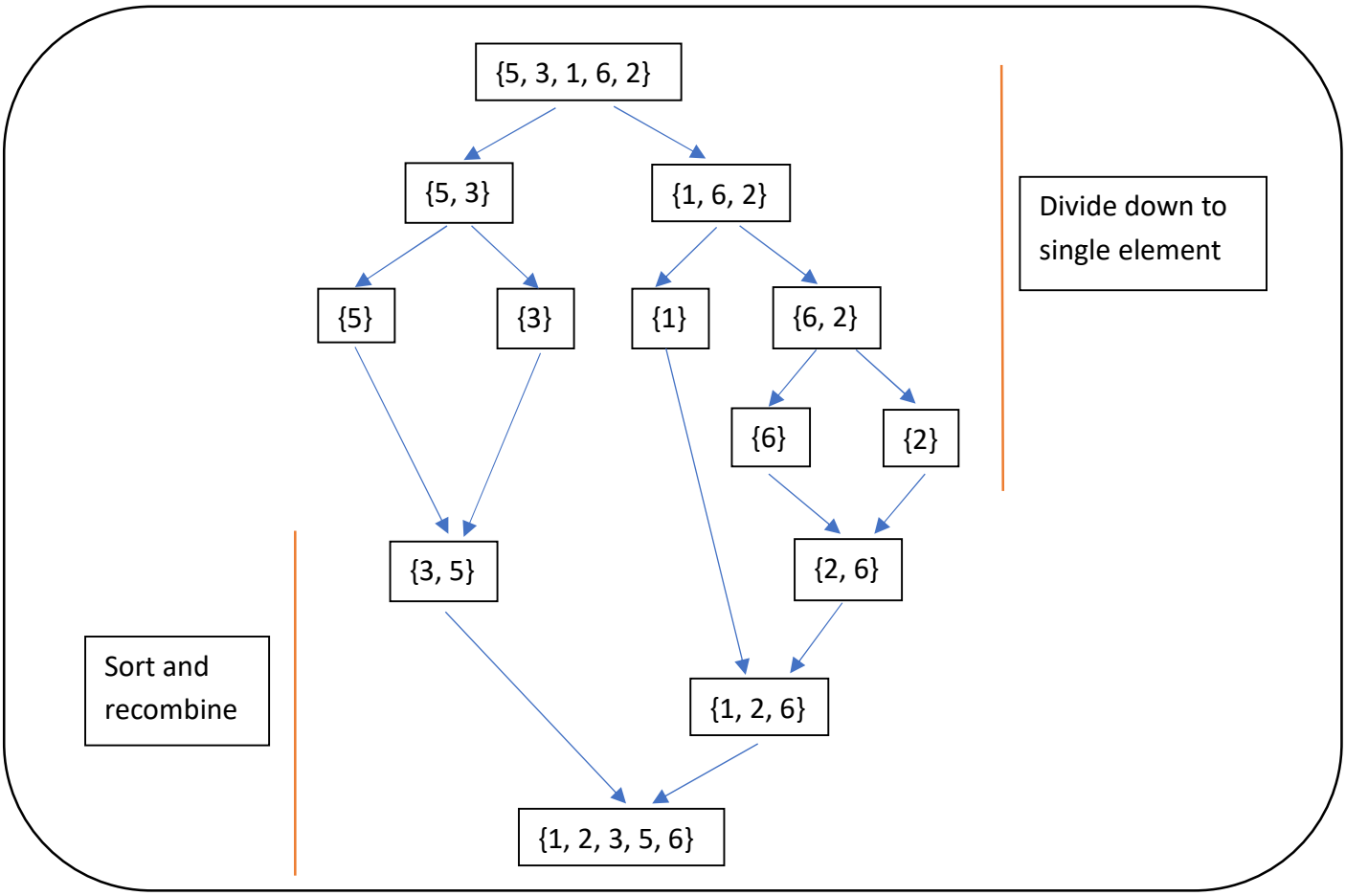
- **Pseudocode**

```
mergeSort (L) {
        if (|L| <= 1)                          B ( Base case)
                return L
        Split L into two non-empty sets L1 and L2   D (Decomposition)
        M1 <- mergeSort (L1)                   C (Calling sub-routines)
        M2 <- mergeSort (L2)

        M <- merge (M1, M2)                    R (Recombination)
        return M
}
merge (M1, M2) {
        if (|M1| = 0)
                return M2
        if (|M2| = 0)
                return M1
        if (x1 <= y1)
                M' <- merge (M1 - {x1}, M2)
                return (x1||M')
        else
                M' <- merge (M1, M2 - {y1})
                return (y1||M')
}
```

Let $M_1 = \{x_1, x_2, x_3, \ldots, x_n\}$
$M_2 = \{y_1, y_2, y_3, \ldots, y_m\}$

- **Recursion Tree Structure**




- **Time Complexity Analysis**

  *T(n) = T(k) + T(n-k) + "merge"* ----(i)

  For the merge section

  *T(n) = T(n-1) + O(1)* -> Solving this we get time complexity as O(n). Combing this with eq. i we get

  *T(n) = T(k) + T(n-k) + O(n)*

  If k = 1 then this solves as **$O(n^2)$**

  If k = n/2 then **O(nlogn)**

- **Iterative merge sort**

  Iterative Merge Sort is a variation of the Merge Sort algorithm that employs a bottom-up, non-recursive approach to sorting. Instead of recursively dividing the array into smaller subarrays, it begins by treating each element as an individual sorted subarray. These subarrays are then merged in pairs to form larger sorted subarrays. This process continues iteratively, doubling the size of the subarrays in each pass, until the entire array is merged into a single sorted sequence. By systematically merging sorted segments, Iterative Merge Sort efficiently organizes the data while maintaining the same time complexity as its recursive counterpart, **O(nlogn).**

## d. Quick Sort

- **Concept and Working principle**

  Quick Sort is a divide-and-conquer sorting algorithm that *sorts an array by partitioning it into smaller subarrays based on a pivot element*. The pivot divides the array into two parts: elements less than the pivot and elements greater than or equal to the pivot. This process is recursively applied to the subarrays until the entire array is sorted.

- **Pseudocode**

```
quickSort (L) {
        if (|L| <= 1)                   B (Base Case)
                return L
        choose random x_i from L
        create L_1 = {x | x < x_i}      D (Decomposition)
        and L_2 = {x | x > x_i}
        M_1 <- quickSort(L_1)           C (Calling sub-routines)
        M_2 <- quickSort(L_2)
        return (M_1 || {x} || M_2)      R (Recombination)
}
```

- **Time Complexity Analysis**

  $T(n) = T(k) + T(n-k) + O(n)$

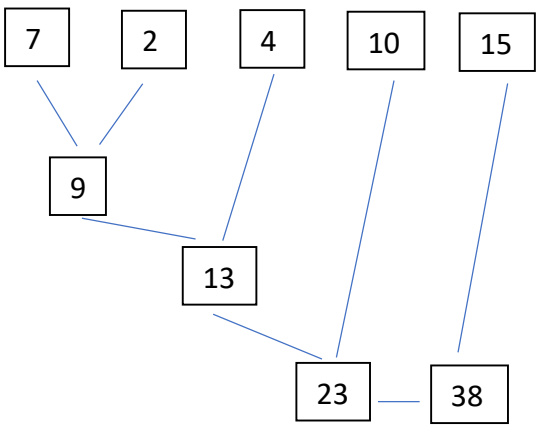
  1. Best case: k = n/2 **O(nlogn)**
  2. Average case: **O(nlogn)**
  3. Worst case: **O($n^2$)**

## Merging N lists

- **Concept and Working Principle**

  We are given n sorted lists and we have to merge all of them. We first try a naïve approach taking any two lists and merging them and merging the rest with the result.

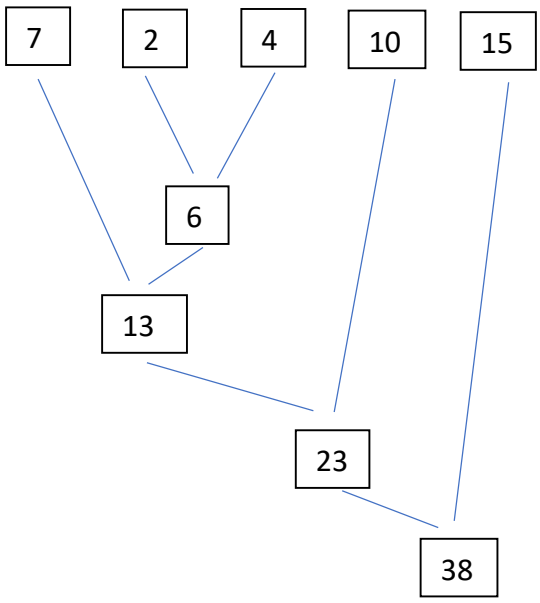
  Counting the number of operations we have to do (merging two array of size m, n take m+n operations)




Thus total no. of operations
9+13+23+38 = 83

Time complexity becomes O($n^2$

However, if we change our order of selecting list and count the total no. of operations




Thus total no. of operations
6+13+23+38 = 80

Therefore, we see that the total number of operations changes if we change our order of choosing the lists.

- **<u>Time complexity Analysis</u>**
  Analysing the costs we get the total time complexity as
  *O(nlogn)+O(klogn)=O(klogn)*