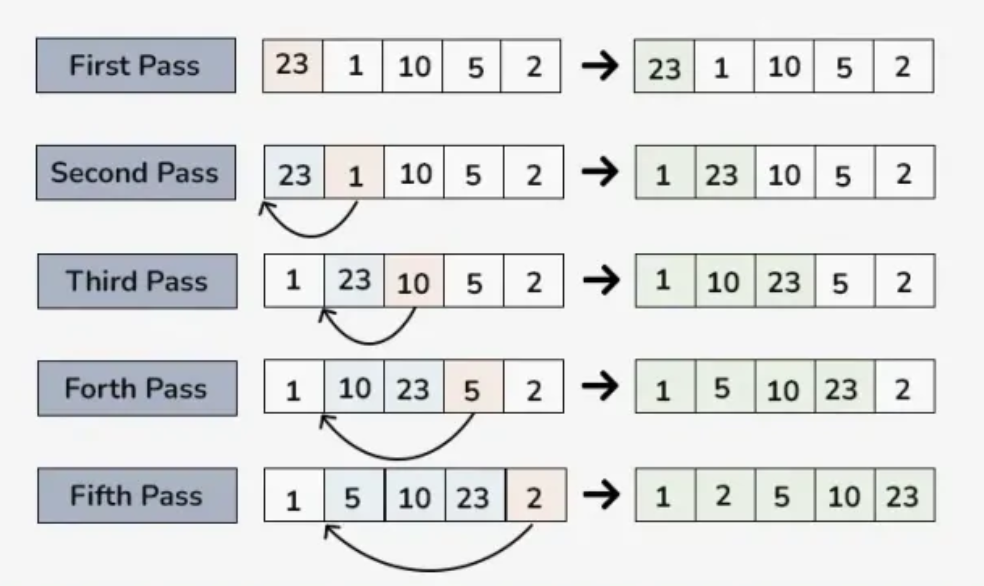
**Chapter 2: SORTING ALGORITHMS**

**Topic – 1: Insertion Sort**

**Algorithm**

* We start from **2nd element** in array.
* All the elements are sorted **along the way** while we pass through each index.
* We start from **second element** because the 1st one is **already** sorted.



* When a smaller element ahead is noticed, we **directly insert** it to the right place.
* It is done by **continuously shifting elements** until we find right place to insert.
* Note that **shifting** & **swapping** are **not** the same thing.

**Runtime**

* Here, we will talk about **how fast** our code will execute.
* Depends directly on the number of input **'n'**.

**T(n) = 'n' runtime input**

* The runtime also depends on the **pattern of input**.
* Runtime for **average case** is often represented as **E(T(n))**.
* **E** means expected & **E(T(n))** means **expected runtime**.
* How fast a program executes also largely depends on the **performance** **of processor**, but we use asymptotic notations to compare quality of codes.

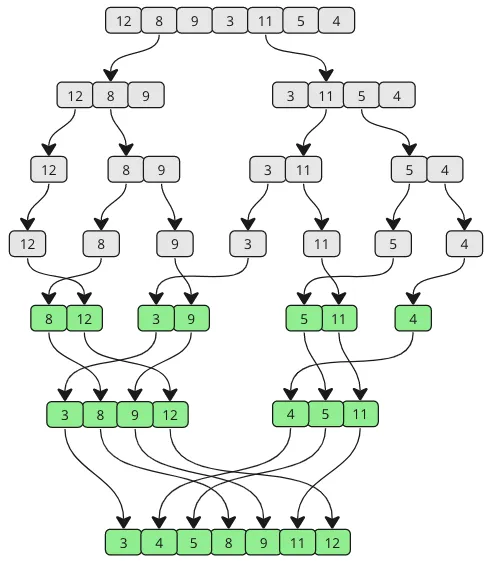
**Algorithm Analysis**

* Insertion sort is an ***in-place sorting*** algorithm.
* **In-place sorting:** Sorting an array without occupying **auxiliary** (extra) **space**.
* Every single part of the code **affects** the duration of execution.
* For examples, **terms compared** in a while loop’s parenthesis.
* **Worst case (θ(n2)) –** When the array is **reverse sorted**.
* **Best case (θ(n)) –** The array is **already sorted**.
* **Average case (θ(n2)) –** **Randomized array** will be sorted.

**Topic – 2: Merge Sort**

**Algorithm**

* It’s a **divide & conquer** technique.
* **Divide –** Dividing the array continuously.
* **Conquer –** Merging the pieces of array thereafter.



**Basic Information**

* Time complexity for **recurrence** of **merge sort** is **θ(n)**.
* This is because we are working on **each element** to be sorted.
* It is **not** an **in-place sorting** algorithm, because elements **can’t** be arranged on the same array using merge sort algorithms.

**Pseudo Code**

**Step 1: Check if number of sub-arrays is 1. [θ(1)]**

**Step 2: Sort the current 2 sub-arrays. [2T(n/2)]**

**Step 3: Create auxiliary array & merge current 2 sub-arrays. [θ(n)]**

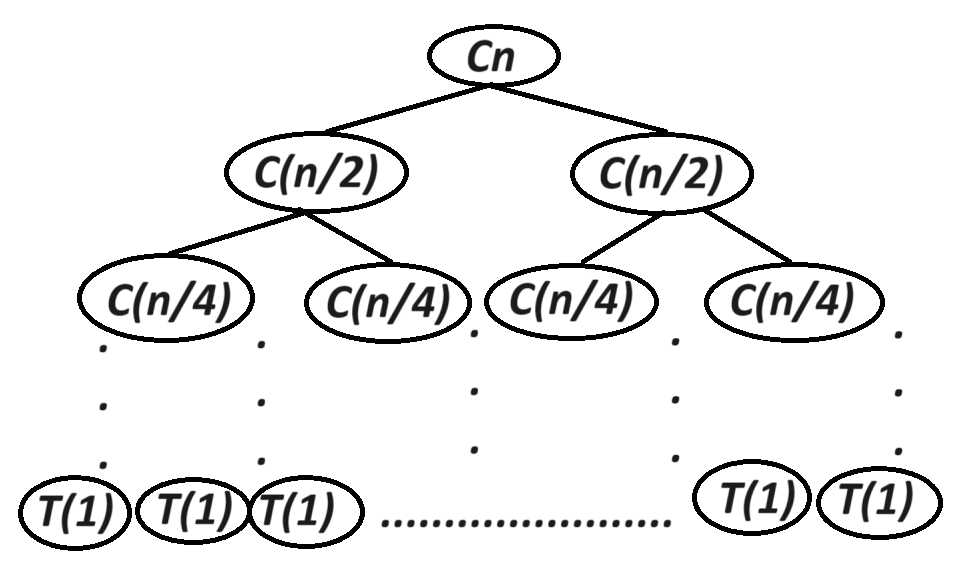
* The reason behind **step 3** having time complexity as **θ(n)** is for arranging **all the elements** in the **new array**.
* We are using **θ** and **T** for separate purposes, **θ** for dealing with **arrays** & **T** for dealing with **elements** in those arrays.
* **θ** and **T are** used as separate part of the equation because all **elements are merged for sure**, but **not** **all are sorted each time**.

**Time Complexity**

**T(n) = θ(1) + 2T(n/2) + θ(n)**

**T(n) = 2T(n/2) + θ(n)**

**Recurrence Tree**



* This diagram clearly shows the **divide & conquer** nature of **merge sort**.
* **T(1)** is the **smallest unit** that can be spent in **any part** of our merge sort algorithm.
* Summing **any row** in figure above provides us the **total time complexity**.
* We can write **T(1)** as **θ(1)** too being more appropriate.
* Means, **n** breaks into **n** parts to provide the smallest unit of value **1**.

**Let height of recurrence tree above be 'h'.**

**No. of elements = 2h = n**

**h = log2(n)**

**T(n) = 2T(n/2) + Cn**

**T(n) = θ(nlog(n))**

* As per our observation, recurrence tree method is **not** a ***full prove method*** due to its inability to provide each detail about the algorithm.
* In future we will discuss ***substitution method*** i.e. a **full prove method**.

**Algorithm Analysis**

* **Worst case (θ(nlog(n)))**
* **Best case (θ(nlog(n)))**
* **Average case (θ(nlog(n)))**