UNIT 2
SEARCHING AND SORTING

Searching

- Searching is the process of finding a given value position in a list of values.
- It decides whether a search key is present in the data or not.
- It is the algorithmic process of finding a particular item in a collection of items.
- There are two types of searching one is Linear Search and another is Binary Search.

Linear Search

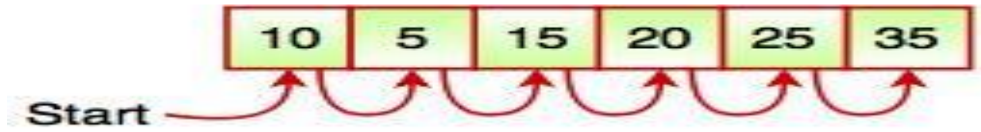
- Sequential search is also called as Linear Search.
- Sequential search starts at the beginning of the list and checks every element of the list.
- It is a basic and simple search algorithm.
- Sequential search compares the element with all the other elements given in the list. If the element is matched, it returns the value index, else it returns -1.
- 

Fig. Sequential Search

Linear Search

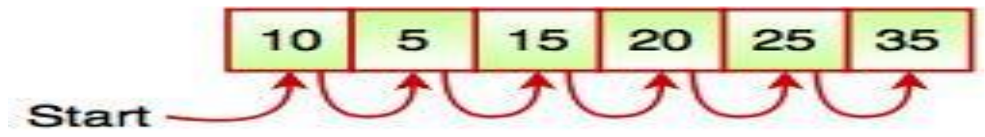


Fig. Sequential Search

The above figure shows how sequential search works.

It searches an element or value from an array till the desired element or value is not found.

If we search the element 25, it will go step by step in a sequence order. It searches in a sequence order.

Sequential search is applied on the unsorted or unordered list when there are fewer elements in a list.

Linear Search

- Linear search is implemented using following steps...
- **Step 1** - Read the search element from the user.
- **Step 2** - Compare the search element with the first element in the list.
- **Step 3** - If both are matched, then display "Given element is found!!!" and terminate the function
- **Step 4** - If both are not matched, then compare search element with the next element in the list.
- **Step 5** - Repeat steps 3 and 4 until search element is compared with last element in the list.
- **Step 6** - If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

Linear Search

- Algorithm For Linear Search

```
LINEAR _SEARCH(A,N,VAL,POS)
Step 1:   [INITIALIZE] SET pos=-1
Step2:   [INITIALIZE] SET I=0
Step3:          Repeat Step 4 while I<N
Step4:               IF A[I] = VAL, then
                          SET POS= I
                          PRINT POS
                          GO to Step 6
               [END OF IF]
          [END OF LOOP]
Step5: PRINT "value Not Present In The Array"
Step6:EXIT
```

Linear search

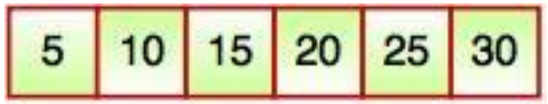
- **Advantages - Linear Search**
- When a key element matches the **first** element in the array, then linear search algorithm is **best** case because executing time of linear search algorithm is 0 (n), where n is the number of elements in an array.
- **Disadvantages - Linear Search**
- Inversely, when a key element matches the **last** element in the array or a key element doesn't matches any element then Linear search algorithm is a **worst** case.
- **Complexity**
- The worst-case and average-case time complexity is O(n).
- $(1+2+\ldots+n)/n = n(n+1)/2n = (n+1)/2$
- The best-case is O(1).

Binary Search

- Binary Search is used for searching an element in a sorted array.
- It is a fast search algorithm with run-time complexity of O(log n).
- Binary search works on the principle of divide and conquer.
- This searching technique looks for a particular element by comparing the middle most element of the collection.
- It is useful when there are large number of elements in an array.

Binary Search

- The below array is sorted in ascending order. As we know binary search is applied on sorted lists only for fast searching.



5	10	15	20	25	30

- **For example,** if searching an element 25 in the 7-element array, following figure shows how binary search works:

Binary Search

- Binary searching starts with middle element. If the element is equal to the element that we are searching then return true. If the element is less than then move to the right of the list or if the element is greater than then move to the left of the list. Repeat this, till you find an element.

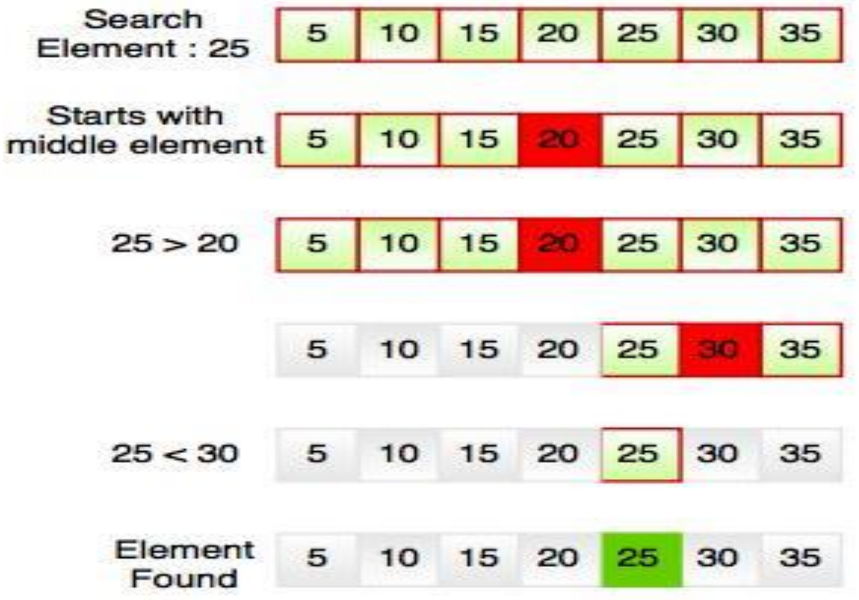


Fig. Working Structure of Binary Search

Binary Search

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Find the middle element in the sorted list.
- **Step 3 -** Compare the search element with the middle element in the sorted list.
- **Step 4 -** If both are matched, then display "Given element is found!!!" and terminate the function.
- **Step 5 -** If both are not matched, then check whether the search element is smaller or larger than the middle element.

Binary Search

- **Step 6 -** If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
- **Step 7 -** If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- **Step 8 -** Repeat the same process until we find the search element in the list or until sublist contains only one element.
- **Step 9 - If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.**

Binary Search

- **Algorithm for Binary Search**
- BINARY SEARCH(A, lower _bound, upper_bound,VAL,POS)
- Step1:[INITIALIZE] SET BEG= lower _bound, END=upper_bound, pos=-1
- Step2: Repeat step3 and step 4 while BEG <= END

```
Step3:            SET  MID=(BEG + END)/2
STEP4:            IF A[MID]=VAL, then
                    POS=MID
                    PRINT POS
                        Go to step  6
               IF A[MID] > VAL  then;
                            SET END =MID-1
```

Binary Search

```
        ELSE
                        SET BEG= MID + 1
                    [END OF IF]
      [END OF LOOP]
```

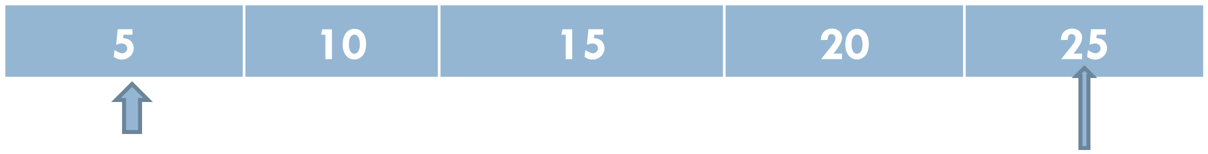
- Step5: IF POS=-1, then

```
      PRINTF"VAL IS NOT PRESENT IN THE ARRAY"
      [END OF IF]
```

- Step6:EXIT

Binary Search

- Example of Binary Search
- https://youtu.be/V_T5NuccwRA
- 0 1 2 3 4

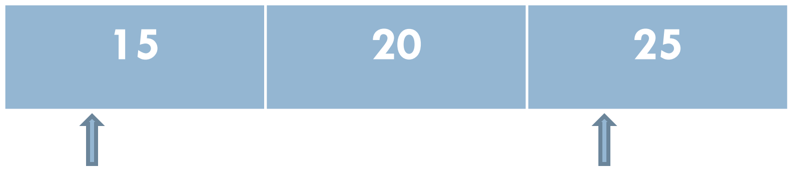


- l r
- Here l=0 , r=4,search data =20
- $\frac{l+r}{2}$ $\frac{0+4 \quad =2}{2}$

so MID value is 15.

Binary Search

- Now there is a three possibilities
- Data==Mid
- Data <Mid
- Data>Mid
- Here is the third possibilities   where search data is greater than mid value. So we have to go through the right hand side.
- 2 3 4



- l r
- $\frac{l+r}{2} \quad \frac{2+4}{2} = 3$ Now, middle term and search term is same.
- i.e. 20

Advantages of Binary Search

- **<u>Advantages</u>**
- It eliminates half of the list from further searching by using the result of each comparison.
- It indicates whether the element being searched is before or after the current position in the list.
- This information is used to narrow the search.
- For large lists of data, it works significantly better than linear search.
-

Disadvantages of Binary Search

- **Disadvantages-**
- The disadvantages of binary search algorithm are-
- **It's more complicated than linear search, and is overkill for very small numbers of elements**.
- It works only on lists that are sorted and kept sorted. That is not always feasible, especially if elements are constantly being added to the list.

Sorting

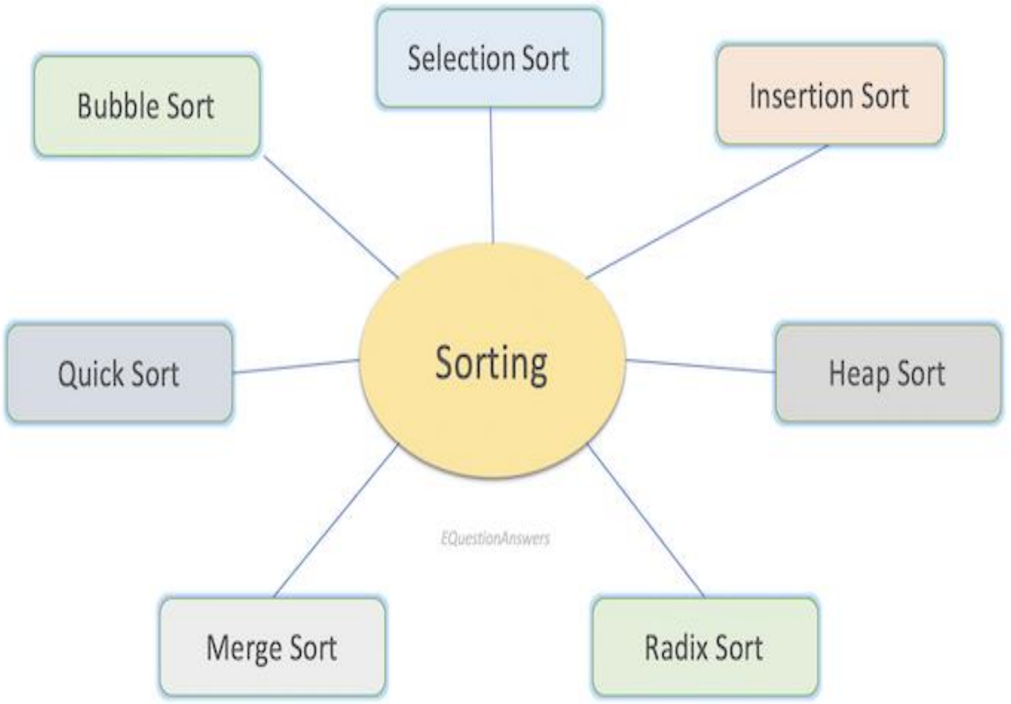
- The arrangement of data in a preferred order is called sorting in the data structure. By sorting data, it is easier to search through it quickly and easily.
- Following are some of the examples of sorting in real-life scenarios –
- **Telephone Directory** – The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- **Dictionary** – The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

Sorting

- There are two different categories in sorting:
- **Internal sorting:** If the input data is such that it can be adjusted in the main memory at once, it is called internal sorting.
- **External sorting:** If the input data is such that it cannot be adjusted in the memory entirely at once, it needs to be stored in a hard disk, floppy disk, or any other storage device. This is called external sorting.

Sorting

- Type of sorting algorithm is given below
- Bubble Sort
- Selection Sort
- Insertion Sort
- Quick Sort
- Radix Sort



Bubble Sort Algorithm

- It is the easiest and simplest of all the sorting algorithms. It works on the principle of repeatedly swapping adjacent elements in case they are not in the right order.

- In simpler terms, if the input is to be sorted in ascending order, the bubble sort will first compare the first two elements in the array. In case the second one is smaller than the first, it will swap the two, and move on to the next element, and so on.

Bubble Sort Algorithm

- Following are the steps involved in bubble sort(for sorting a given array in ascending order):
- Starting with the first element(index = 0), compare the current element with the next element of the array.
- If the current element is greater than the next element of the array, swap them.
- If the current element is less than the next element, move to the next element. **Repeat Step 1**.

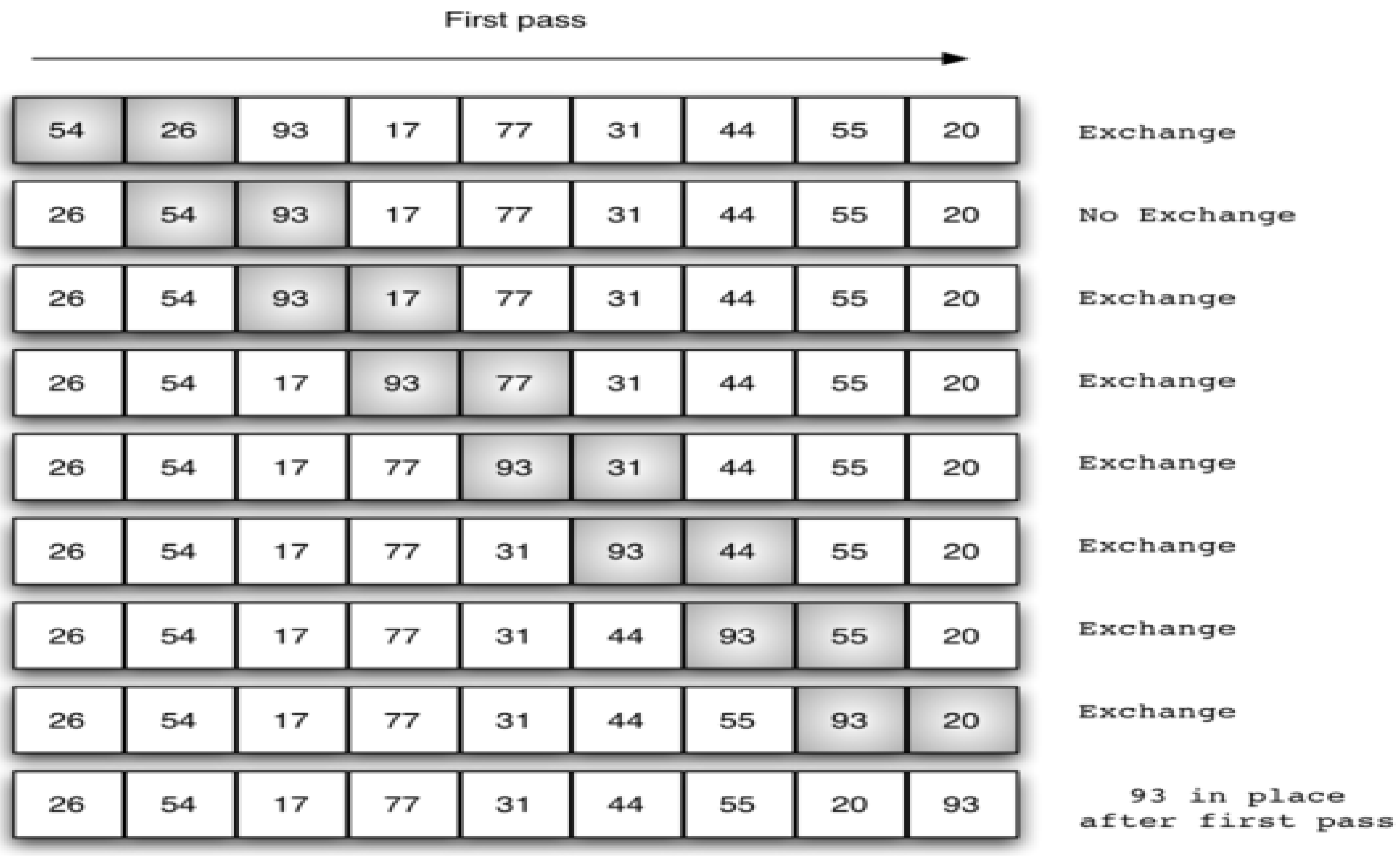
Bubble Sort

- Algorithm for bubble sort

```
Step1:Repeat steps 2 for I=0 toN-1
Step2:          repeat for J=0 to N-I
Step3:              If A[J]>a[J+1],then
                     SWAP A[J] and A[J+1]
              [End OF Inner Loop]
         [End OF Outer Loop]
Step 4:EXIT
```

Bubble Sort

- Example of Bubble Sort



54	26	93	17	77	31	44	55	20	Exchange
26	54	93	17	77	31	44	55	20	No Exchange
26	54	93	17	77	31	44	55	20	Exchange
26	54	17	93	77	31	44	55	20	Exchange
26	54	17	77	93	31	44	55	20	Exchange
26	54	17	77	31	93	44	55	20	Exchange
26	54	17	77	31	44	93	55	20	Exchange
26	54	17	77	31	44	55	93	20	Exchange
26	54	17	77	31	44	55	20	93	93 in place after first pass

Bubble Sort Algorithm

- Example of Bubble sort
- **Pass1:**

15	16	8	6	5

- Compare 15 with 16.Here 15 is less than 16so No swap it.

15	16	8	6	5

15	8	16	6	5
15	8	6	16	5

15	8	6	5	16

Bubble Sort Algorithm

- Pass2:

15	8	6	5	16
8	15	6	5	16
8	6	15	5	16
8	6	5	15	16

- Pass3:

8	6	5	15	16
6	8	5	15	16
6	5	8	15	16
6	5	8	15	16

Bubble Sort Algorithm

- Pass 4:

6	5	8	15	16

5	6	8	15	16

5	6	8	15	16

5	6	8	15	16

- Here N-1 passes is required.
- Here N= No. of elements in array.
- 5-1=4 .so 4 passes are required here .

Advantages of Bubble Sort

- One of the main advantages of a bubble sort is that **it is a very simple algorithm to describe to a computer**.
- There is only really one task to perform (compare two values and, if needed, swap them). This makes for a very small and simple computer program .

Disadvantages of Bubble Sort

The main disadvantage of the bubble sort method is **the time it requires**. It is highly inefficient for large data sets. Additionally, the presence of turtles can severely slow the sort.

Selection Sort Algorithm

- Selection sort is conceptually the most simplest sorting algorithm.
- This algorithm will first find the **smallest** element in the array and swap it with the element in the **first** position, then it will find the **second smallest** element and swap it with the element in the **second** position, and it will keep on doing this until the entire array is sorted.
- It is called selection sort because it repeatedly **selects** the next-smallest element and swaps it into the right place.

Selection Sort Algorithm

- **Step by Step Process**
- The selection sort algorithm is performed using the following steps...
- **Step 1 -** Select the first element of the list (i.e., Element at first position in the list).
- **Step 2:** Compare the selected element with all the other elements in the list.
- **Step 3:** In every comparison, if any element is found smaller than the selected element (for Ascending order), then both are swapped.
- **Step 4:** Repeat the same procedure with element in the next position in the list till the entire list is sorted.

Selection Sort Algorithm

Algorithm for Selection Sort

```
SMALLEST (ARR,K,N,POS)
Step1:[Initialize]SET SMALL =ARR[K]
Step2: :[Initialize]SET POS=K
Step3:Repeat for J=K+1 to N
              IF SMALL >ARR[J],then
                    SET SMALL =ARR[J]
                     SET POS=J
           [END OF IF]
            [END OF LOOP]
   Step 4:Exit
```

Selection Sort Algorithm

- **Sub Algorithm:**
- SELECTION SORT(ARR,N)
- Step1:Repeat Step 2 and 3 for k=1 to N-1
- Step2: CALL SMALLEST (ARR,K,N,POS)
- Step3: SWAP A[K] with ARR[POS]
- [END OF LOOP]
- Step4:Exit

Selection Sort Algorithm

- Example of Selection Sort Algorithm

7	4	10	8	3	1

- Sorted Unsorted list
- First find the smallest element then swap it from 0th index.
- Pass 1:

1	4	10	8	3	7

- Pass2:

1	3	10	8	4	7

- Pass3:

1	3	4	8	10	7

- Pass4:

1	3	4	7	10	8

- Pass5:

1	3	4	7	8	10

Insertion Sort Algorithm

- Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands.
- The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.
- **Insertion sort** is the sorting mechanism where the **sorted** array is built having one item at a time. The array elements are compared with each other sequentially and then arranged simultaneously in some particular order.

Insertion Sort Algorithm

- **Step by Step Process**
- The insertion sort algorithm is performed using the following steps...
- **Step 1 -** Assume that first element in the list is in sorted portion and all the remaining elements are in unsorted portion.
- **Step 2:** Take first element from the unsorted portion and insert that element into the sorted portion in the order specified.
- **Step 3:** Repeat the above process until all the elements from the unsorted portion are moved into the sorted portion.

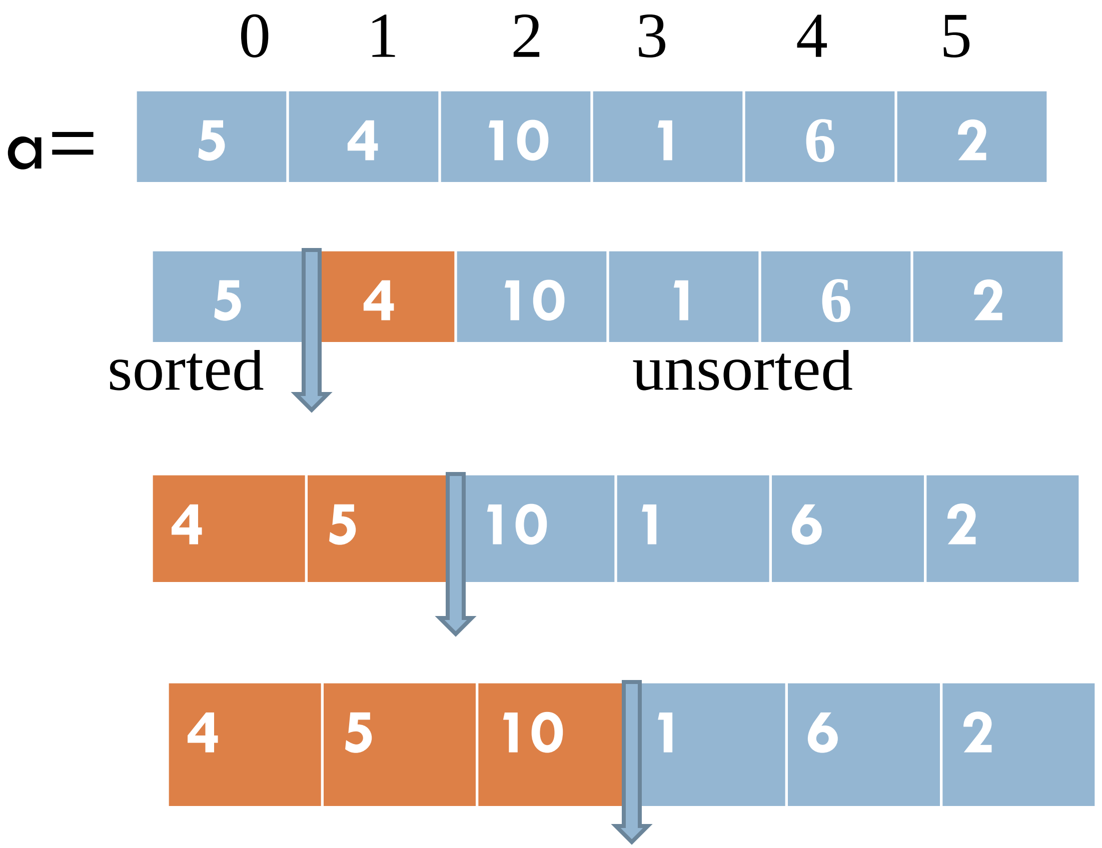
Insertion Sort Algorithm

- Algorithm

```
Step1:Repeat step2 to 5 for k=1 to N
Step2:        SET TEMP=ARR[K]
Step3:        SET J=K-1
Step4:        Repeat while TEMP<=ARR[J]
                  SET ARR[J+1]=ARR[J]
                  SET J=J-1
              [END OF INNER LOOP]
Step5:        SET ARR[J+1]=TEMP
      [END OF LOOP]
Step6:EXIT
```

Insertion Sort Algorithm

- Example n=6



temp=4

temp=10

temp=1

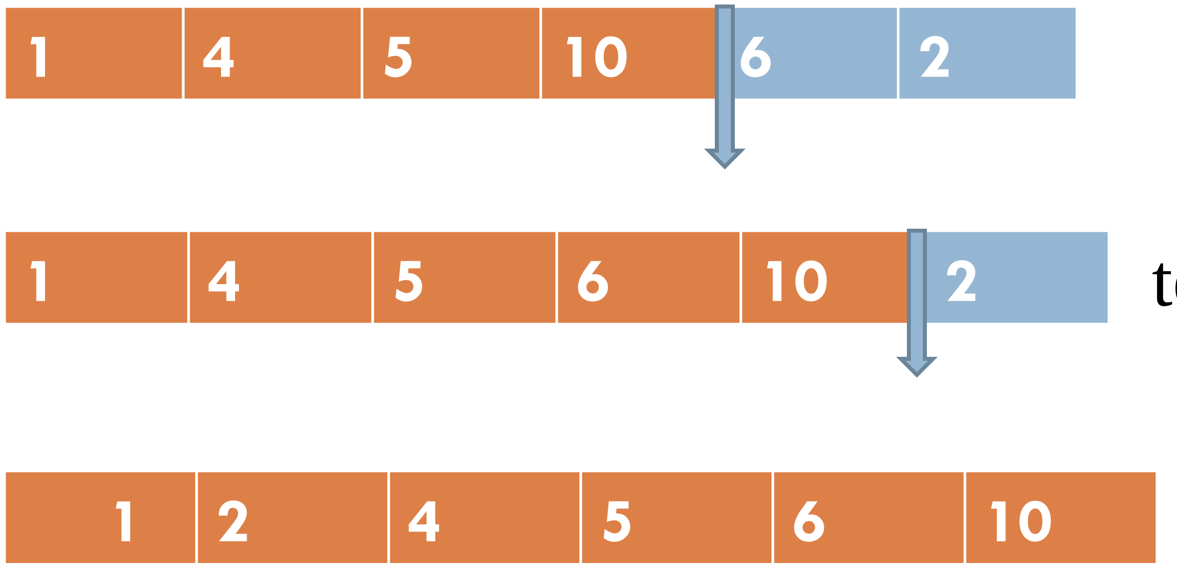
Insertion Sort Algorithm

- Example

temp=6

-
-

temp=2



Quick Sort Algorithm

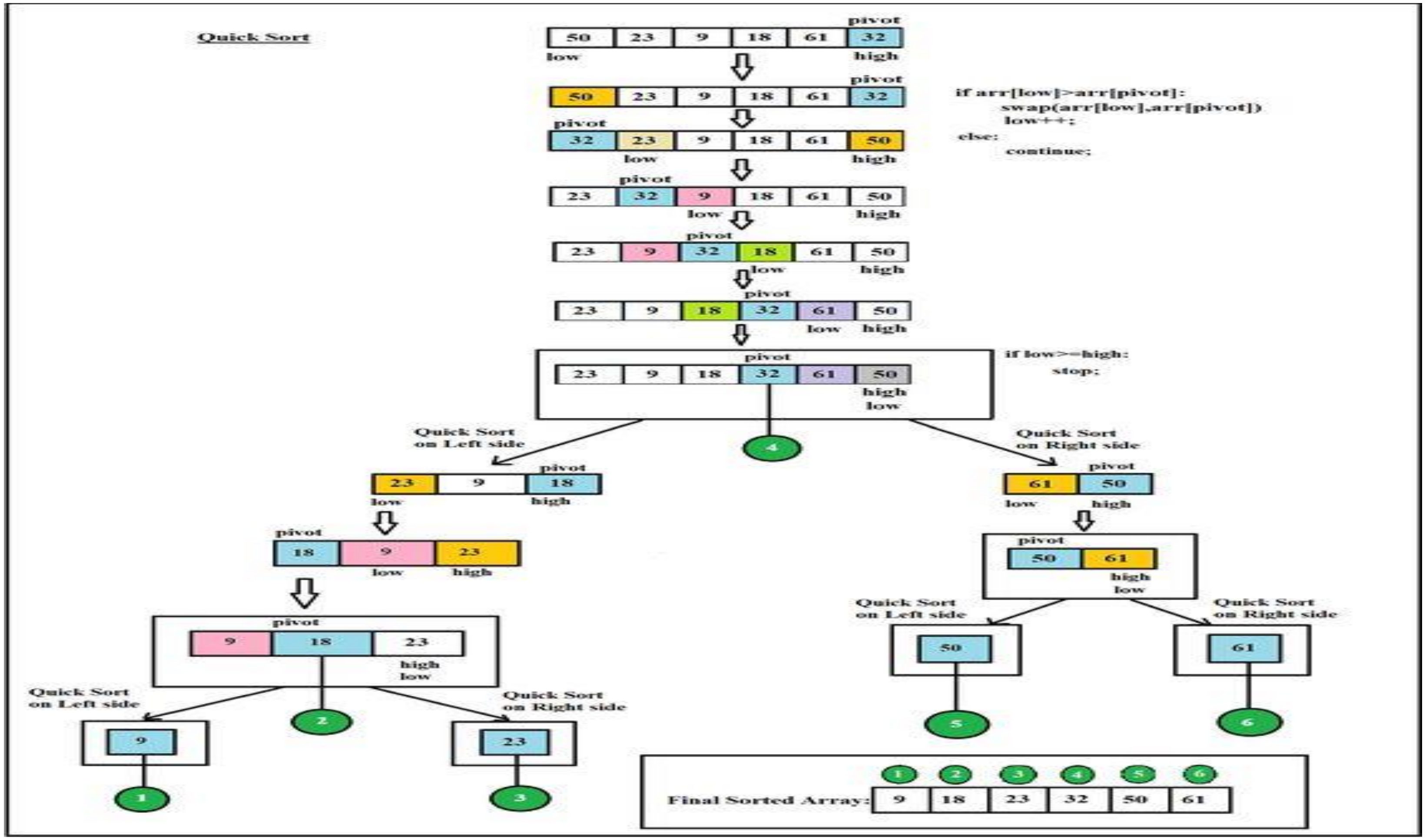
- Quick sort is a fast sorting algorithm used to sort a list of elements. Quick sort algorithm is invented by **C. A. R. Hoare**.
- The quick sort algorithm attempts to separate the list of elements into two parts and then sort each part recursively. That means it use **divide and conquer** strategy.
- In quick sort, the partition of the list is performed based on the element called ***pivot***. Here pivot element is one of the elements in the list.
- The list is divided into two partitions such that "**all elements to the left of pivot are smaller than the pivot and all elements to the right of pivot are greater than or equal to the pivot**".

Quick Sort Algorithm

- **Step by Step Process**
- In Quick sort algorithm, partitioning of the list is performed using following steps...
- **Step 1 -** Consider the first element of the list as **pivot** (i.e., Element at first position in the list).
- **Step 2 -** Define two variables i and j. Set i and j to first and last elements of the list respectively.
- **Step 3 -** Increment i until list[i] > pivot then stop.
- **Step 4 -** Decrement j until list[j] < pivot then stop.
- **Step 5 -** If i < j then exchange list[i] and list[j].
- **Step 6 -** Repeat steps 3,4 & 5 until i > j.
- **Step 7 -** Exchange the pivot element with list[j] element.

Quick Sort Algorithm

- Example



```
if arr[low]>arr[pivot]:
        swap(arr[low],arr[pivot])
        low++;
else:
        continue;
```

```
if low>=high:
        stop;
```

Radix Sort Algorithm

- **Radix sort** is a small method that many people intuitively use when alphabetizing a large list of names. Specifically, the list of names is first sorted according to the first letter of each name, that is, the names are arranged in 26 classes.
- Radix sort works counter-intuitively by sorting on the least significant digits first. On the first pass, all the numbers are sorted on the least significant digit and combined in an array. Then on the second pass, the entire numbers are sorted again on the second least significant digits and combined in an array and so on.
- Radix sort also called as **Bucket sort**.

Radix Sort Algorithm

- **Step by Step Process**
- The Radix sort algorithm is performed using the following steps...
- **Step 1 -** Define 10 queues each representing a bucket for each digit from 0 to 9.
- **Step 2 -** Consider the least significant digit of each number in the list which is to be sorted.
- **Step 3 -** Insert each number into their respective queue based on the least significant digit.
- **Step 4 -** Group all the numbers from queue 0 to queue 9 in the order they have inserted into their respective queues.
- **Step 5 -** Repeat from step 3 based on the next least significant digit.
- **Step 6 -** Repeat from step 2 until all the numbers are grouped based on the most significant digit.

Radix Sort Algorithm

- Algorithm:
- Step1:Find the largest number in ARR as LARGE
- Step2:[Initialize]SET NOP=Number of digits LARGE
- Step3:SET PASS=0
- Step4:Repeat step 5 while pass <=NOP-1
- Step5:   SET I=0 AND Initialize Bucket
- Step6:   Repeat step 7 to Step 9 while I<N-1
- Step7:   SET DIGIT=digit at passth place in A[I]
- Step8:   Add A[I] to the bucket numbered DIGIT
- Step9:   INCREMENT bucket count for bucket numbered DIGIT
- Step10:Collect the numbers in the bucket
- [END OF LOOP]
- Step11:END

Radix Sort Algorithm

- Example: Sort the given no .using Radix sort
- 345,654,924,123,567,472,555,808,911.
- Pass1:

No.	0	1	2	3	4	5	6	7	8	9
345						345				
654					654					
924					924					
123				123						
567								567		
472			472							
555						555				
808									808	
911		911								

Radix Sort Algorithm

- **Pass2:** 911,472,123,654,924,345,555,567,808

No.	0	1	2	3	4	5	6	7	8	9
911		911								
472								472		
123			123							
654						654				
924			924							
345					345					
555						555				
567							567			
808	808									

Radix Sort Algorithm

- Pass3:808,911,123,924,345,654,555,567,472

No.	0	1	2	3	4	5	6	7	8	9
808									808	
911										911
123		123								
924										924
345				345						
654							654			
555						555				
567						567				
472					472					

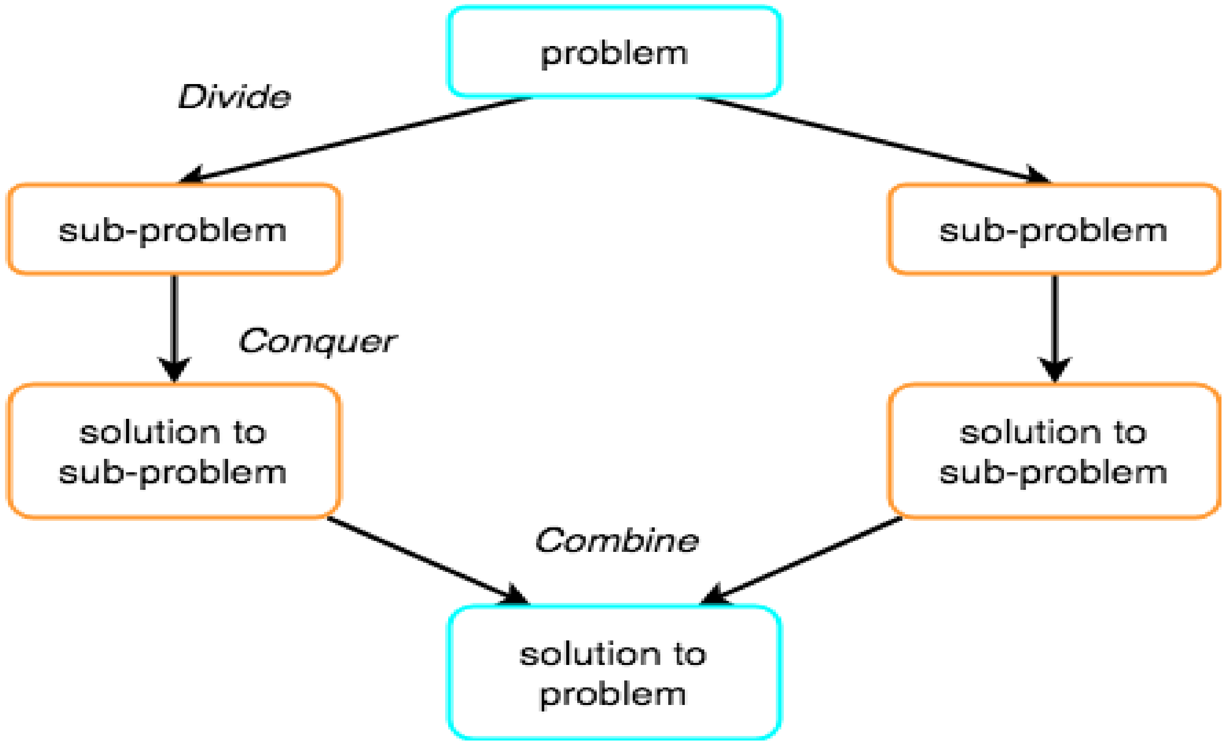
- Sorted list is123,345,472,555,567,654,808,911,924

Merge Sort Algorithm

- In **Merge Sort**, the given unsorted array with n elements, is divided into n subarrays, each having **one** element, because a single element is always sorted in itself. Then, it repeatedly merges these subarrays, to produce new sorted subarrays, and in the end, one complete sorted array is produced.
- The concept of Divide and Conquer involves three steps:
- **Divide** the problem into multiple small problems.
- **Conquer** the sub problems by solving them. The idea is to break down the problem into atomic sub problems, where they are actually solved.
- **Combine** the solutions of the sub problems to find the solution of the actual problem.

Merge Sort Algorithm

- Flow of Merge Sort Algorithm



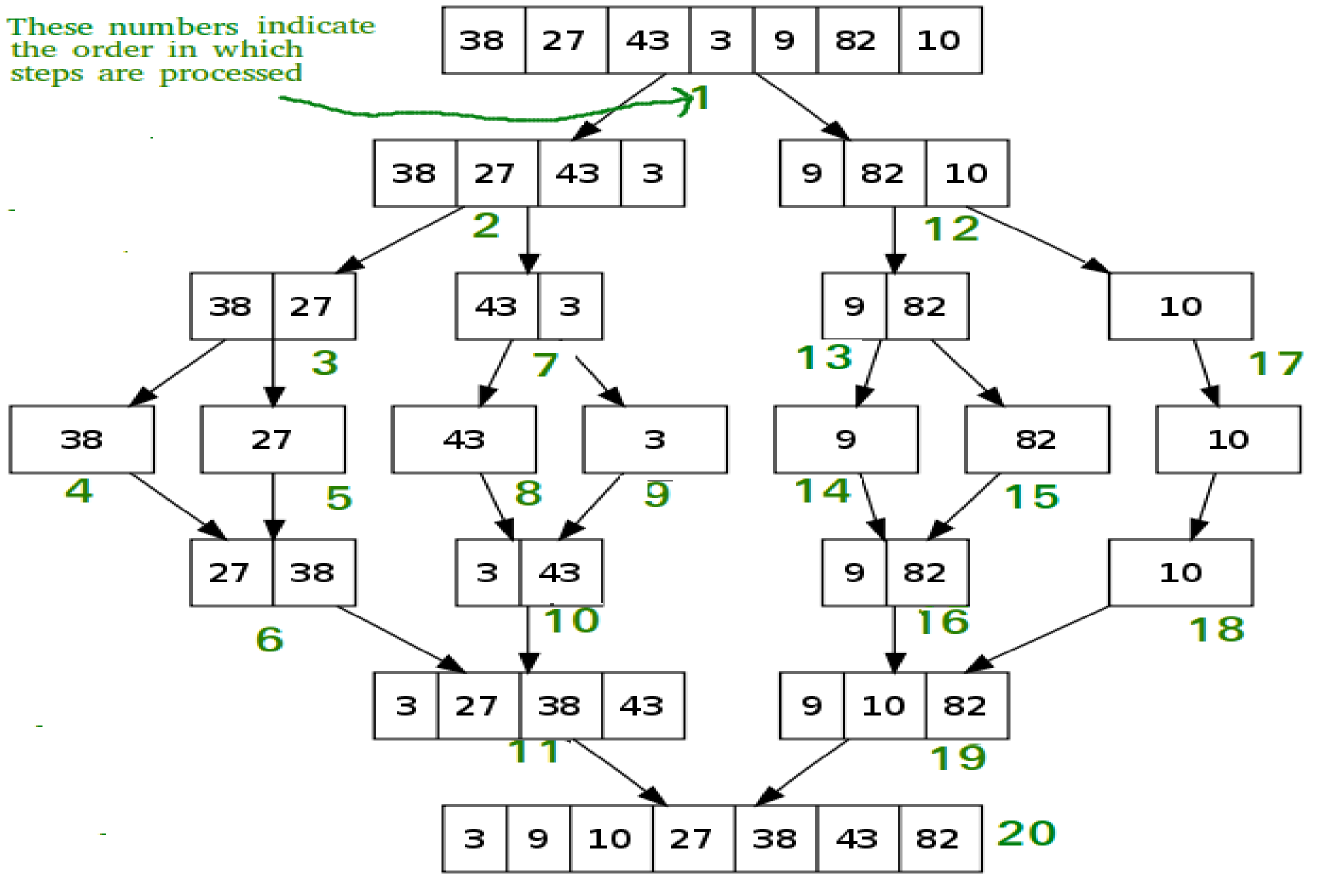
Merge Sort Algorithm

- Algorithm
- **MergeSort(arr[], l, r)**
- If r > l
- **1.** Find the middle point to divide the array into two halves: middle m = (l+r)/2
- **2.** Call mergeSort for first half: Call mergeSort(arr, l, m)
- **3.** Call mergeSort for second half: Call mergeSort(arr, m+1, r)
- **4.** Merge the two halves sorted in step 2 and 3:

  Call merge(arr, l, m, r)

Merge Sort Algorithm

- Example



Performance Analysis

- if we want to go from city "A" to city "B", there can be many ways of doing this. We can go by flight, by bus, by train and also by bicycle.
- Depending on the availability and convenience, we choose the one which suits us. Similarly, in computer science, there are multiple algorithms to solve a problem. When we have more than one algorithm to solve a problem, we need to select the best one.
- Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem. When there are multiple alternative algorithms to solve a problem, we analyze them and pick the one which is best suitable for our requirements.

Performance Analysis

- Formal definition is given below…………….
- Performance of an algorithm is a process of making evaluative judgment about algorithms.
- Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.
- Here, resources means memory required by that algorithm, the execution speed of that algorithm, easy to understand, easy to implement.
- Performance analysis of an algorithm is the process of calculating space and time required by that algorithm.

Space Complexity

- When we design algorithm to solve problem ,it needs some computer memory to complete it's execution.
- for any algorithm, memory is required for the following purpose.
- To store program instructions.
- To store constant values.
- To store variable values.
- Space complexity of an algorithm can be defined as <u>total amount of computer memory required by an algorithm to complete its execution is called as space complexity.</u>

Time Complexity

- Every algorithm requires some amount of computer time to executes its instruction to perform the task. This computer time required is called time complexity.
- The time complexity of an algorithm can be defined as follows...
- The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

Asymptotic Notation

- Whenever we want to perform analysis of an algorithm, we need to calculate the complexity of that algorithm. But when we calculate the complexity of an algorithm it does not provide the exact amount of resource required.
- So instead of taking the exact amount of resource, we represent that complexity in a general form (Notation) which produces the basic nature of that algorithm. We use that general form (Notation) for analysis process.
- **Asymptotic notation of an algorithm is a mathematical representation of its complexity.**
- Majorly, we use THREE types of Asymptotic Notations and those are as follows...
- Big - Oh (O)
- Big - Omega (Ω)
- Big - Theta (Θ)

Big - Oh Notation (O)

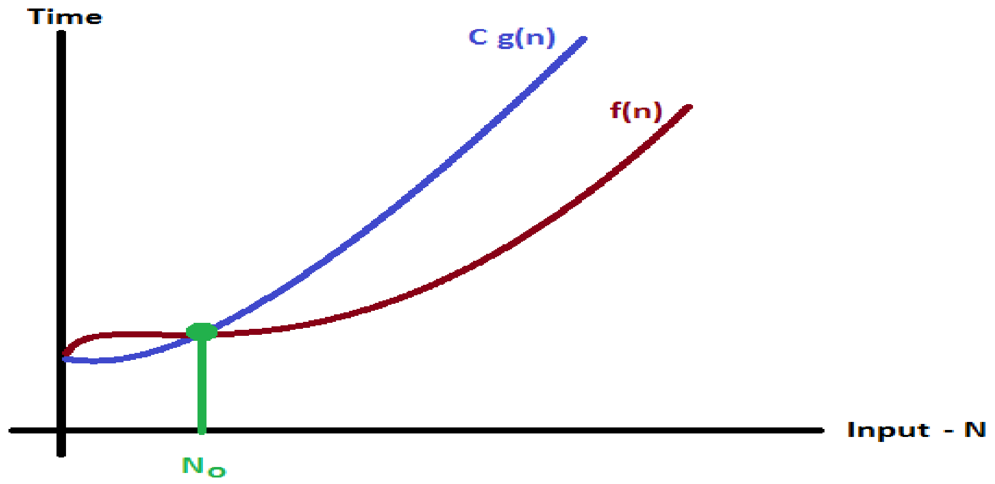
- Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity. That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.
- Big – Oh Notation can be defined as follows.

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}.$$

- **f(n) = O(g(n))**

Big - Oh Notation (O)

- Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



- In above graph after a particular input value n_0, always C g(n) is greater than f(n) which indicates the algorithm's upper bound.

Big - Oh Notation (O)

- **Example**
- Consider the following f(n) and g(n)...
  f(n) = 3n + 2
  g(n) = n
  If we want to represent **f(n)** as **O(g(n))** then it must satisfy **f(n) <= C g(n)** for all values of **C > 0** and $\mathbf{n_0}$**>= 1**
  f(n) <= C g(n)
  ⇒3n + 2 <= C n
  Above condition is always TRUE for all values of **C = 4** and **n >= 2**.
  By using Big - Oh notation we can represent the time complexity as follows...
  3n + 2 = O(n)

Big - Omega Notation (Ω)

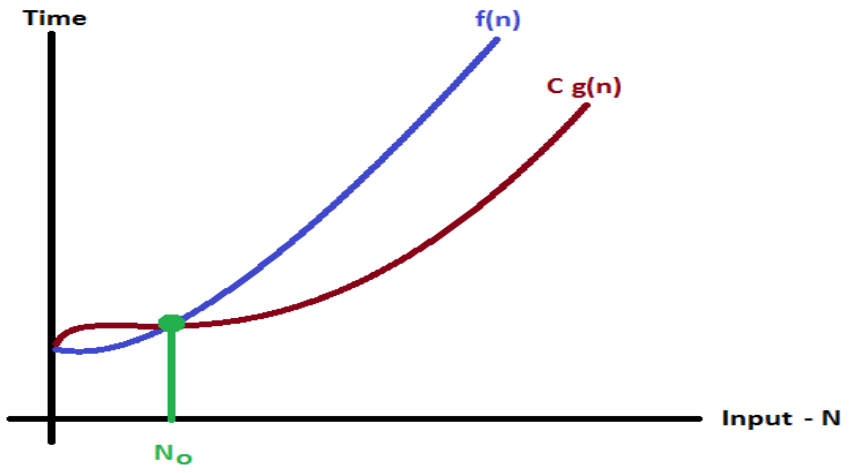
- Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity. That means Big-Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big-Omega notation describes the best case of an algorithm time complexity.
  Big - Omega Notation can be defined as follows...

$$\Omega(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}.$$

- **f(n) = Ω(g(n))**

Big - Omega Notation (Ω)

- Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis.



In above graph after a particular input value n_0, always C g(n) is less than f(n) which indicates the algorithm's lower bound.

Big - Omega Notation (Ω)

- **Example**
- Consider the following f(n) and g(n)...

  $f(n) = 3n + 2$

  $g(n) = n$

  If we want to represent **f(n)** as **Ω(g(n))** then it must satisfy **f(n) >= C g(n)** for all values of **$C > 0$** and **$n_0 >= 1$**

  $f(n) >= C\, g(n)$

  $\Rightarrow 3n + 2 >= C\, n$

  Above condition is always TRUE for all values of

  C = 1 and **n >= 1**.

  By using Big - Omega notation we can represent the time complexity as follows...

  $3n + 2 = \Omega(n)$

Big - Theta Notation (Θ)

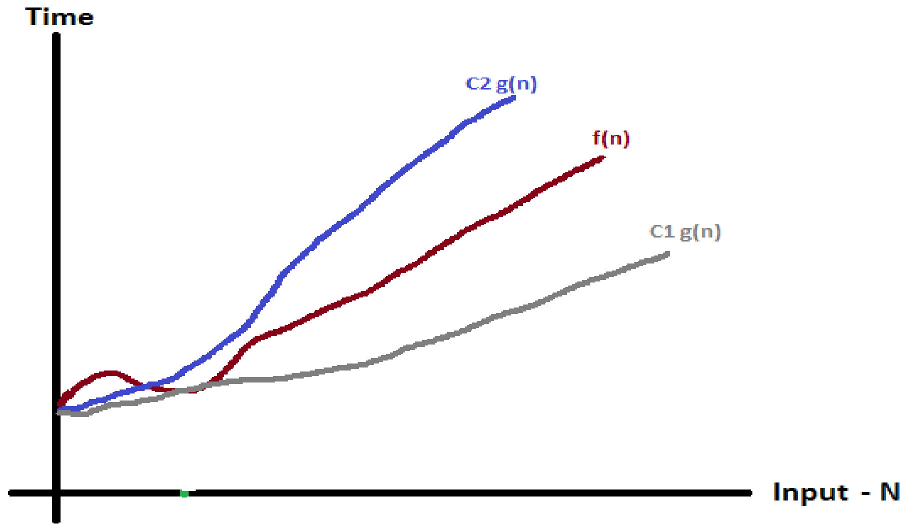
- Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity. That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity. Big - Theta Notation can be defined as follows...

$$\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \} .$$

- **f(n) = Θ(g(n))**

Big - Theta Notation (Θ)

- Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis.



- In above graph after a particular input value n_0, always C_1 g(n) is less than f(n) and C_2 g(n) is greater than f(n) which indicates the algorithm's average bound.

Big - Theta Notation (Θ)

- **Example**
- Consider the following f(n) and g(n)...

  $f(n) = 3n + 2$

  $g(n) = n$

  If we want to represent **f(n)** as **Θ(g(n))** then it must satisfy **$C_1\ g(n) <= f(n) <= C_2\ g(n)$** for all values of **$C_1 > 0$, $C_2 > 0$** and **$n_0 >= 1$**

  $C_1\ g(n) <= f(n) <= C_2\ g(n)$

  $\Rightarrow C_1\ n <= 3n + 2 <= C_2\ n$

  Above condition is always TRUE for all values of

  $C_1 = 1$, $C_2 = 4$ and **$n >= 2$**.

  By using Big - Theta notation we can represent the time complexity as follows...

  $3n + 2 = \Theta(n)$

Comparison of Sorting Method

- The comparison of sorting methods is performed based on the **Time complexity** and **Space complexity** of sorting methods.

Sorting Method	Time Complexity Worst Case	Time Complexity Average Case	Time Complexity Best Case	Space Complexity
Bubble Sort	$n(n-1)/2 = O(n^2)$	$n(n-1)/2 = O(n^2)$	$n(n-1)/2 = O(n^2)$	Constant
Insertion Sort	$n(n-1)/2 = O(n^2)$	$n(n-1)/4 = O(n^2)$	$O(n)$	Constant
Selection Sort	$n(n-1)/2 = O(n^2)$	$n(n-1)/2 = O(n^2)$	$n(n-1)/2 = O(n^2)$	Constant
Quick Sort	$n(n+3)/2 = O(n^2)$	$O(n \log n)$	$O(n \log n)$	Constant
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Constant
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Depends