
Data Structure

PreCAT Scope : Data Structures

- In Section B => **7 Questions** are there for this subject

and mostly all questions are **theory based or concepts / pseudocode oriented.**

- In this course the main focus is on **implementation of basic data structures** and **introduction to an advanced data structures** to build a base which is required to learn and implement advanced data structures and algorithms in CDAC courses.

Data Structures: Introduction

+ Introduction

- Data structure
- Algorithm and analysis of an algorithm

+ Array

- Concept & definition
- **Searching Algorithms:**

1. Linear Search (Algorithm & Implementation)
2. Binary Search (Algorithm & Implementation)

- Sorting Algorithms:

1. Selection Sort (Algorithm & Implementation)
2. Bubble Sort (Algorithm & Implementation)
3. Insertion Sort (Algorithm & Implementation)
4. Quick Sort (Only Algorithm)
5. Merge sort (Only Algorithm)

Data Structures: Introduction

+ Linked List

- Concept & definition
- Types of Linked List
- Operations on Linked List : addition, deletion & traversal.
- Difference between an array and linked list.

+ Stack

- Concept & definition
- Implementation of stack data structure (by using an array)
- Stack applications algorithms:

1. Conversion of infix expression into its equivalent prefix
2. Conversion of infix expression into its equivalent postfix
3. Conversion of prefix expression into its equivalent postfix
4. Postfix expression evalution

Data Structures: Introduction

+ Queue

- Concept & definition
- Types of queue
- Implementation of linear queue & circular queue
- Applications of queue data structure

+ Introduction to an advanced data structure

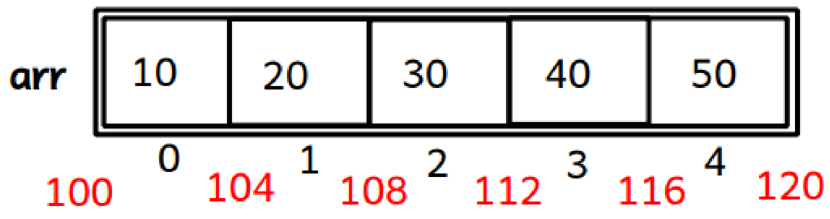
- Tree : terminologies
- Graph : terminologies
- Hash Table

Q. What is Data Structure?

- Data Structure is **a way to store data elements into the memory (i.e. into the main memory) in an organized manner** so that operations like **addition, deletion, traversal, searching, sorting** etc... can be performed on it efficiently.

Data Structures: Introduction

 # Array: It is a **basic/linear data structure**, which is a collection/list of **logically related similar type of data elements,** gets stored into the memory at **contiguos locations.**



arr: int [ ] --> arr is an array variable which is a collection/list of 5 int elements

arr = 100 - array name itself is a base address of block of 20 bytes.

arr[0] : int --> arr[0] = 10, & arr[0] = 100

arr[1] : int --> arr[1] = 20, & arr[1] = 104

arr[2] : int --> arr[2] = 30, & arr[2] = 108

arr[3] : int --> arr[3] = 40, & arr[3] = 112

arr[4] : int --> arr[4] = 50, & arr[4] = 116

arr[0] = *(arr+0) = *(100+0) = *(100) = 10

arr[1] = *(arr+1) = *(100+1) = *(104) = 20

arr[2] = *(arr+2) = *(100+2) = *(108) = 30

arr[3] = *(arr+3) = *(100+3) = *(112) = 40

arr[4] = *(arr+4) = *(100+4) = *(116) = 50

Data Structures: Introduction

 # Structure: It is a **basic/linear data structure**, which is a collection/list of **logically related similar and disimmilar type of data elements,** gets stored into the memory collectively as a **single entity** (as a single record).

```
struct employee {
        int empid;//4 bytes
        char name[32];//32 bytes
        float salary;//4 bytes
};
```

emp

111	"sachin pawar"	9999.99
empid	name	salary

100 104 136 140

struct employee emp;

sizeof structure = sum of size of all its members ==> sizeof(struct student) = 40 bytes

- We can access members of the structure by its variable through dot operator **(.)**

emp: struct employee

emp.empid: int

emp.name : char [ ]

emp.salary: float

- We can access members of the structure by its pointer variable through an arrow operator **(->)**

struct employee *pe = &emp;

pe: struct student * -> sizeof(pe) = 4 bytes

pe->empid: int

pe->name : char [ ]

pe->salary: float

Data Structures: Introduction

Two types of **Data Structures** are there:

1. Linear / Basic Data Structures: data elements gets stored into the memory in a linear manner (e.g. sequentially) and hence can be accessed **linearly / sequentially.**

- Array
- Structure & Union
- Linked List
- Stack
- Queue

2. Non-linear / Advanced Data Structures: data elements gets stored into the memory in a **non-linear manner (e.g. hierarchical)** and hence can be accessed **non-linearly.**

- Tree (Hierarchical)
- Graph
- Hash Table

Data Structures: Introduction

Q. What is a Program?

- A Program is a finite set of instructions written in any programming language (like C, C++, Java, Python, Assembly etc...) given to the machine to do specific task.

Q. What is an Algorithm?

- An algorithm is a finite set of instructions written in human understandable language (like english), if followed, acomplishesh a given task.

Q. What is a Pseudocode?

- An algorithm is a finite set of instructions written in human understandable language (like english) **with some programming constraints**, if followed, acomplishesh a given task, such an algorithm also called as **pseudocode.**

- An algorithm is a template whereas a program is an implementation of an algorithm.

Data Structures: Introduction

Example: An algorithm to do sum of all array elements

```
Algorithm ArraySum(A, n)//whereas A is an array of size n
{
    sum=0;//initially sum is 0
    for( index = 1 ; index <= size ; index++ ) {
    sum += A[ index ];//add each array element into the sum
    }
    return sum;
}
```

- In this algorithm, **traversal/scanning** operation is applied on an array. Initially sum is 0, each array element gets added into to the sum by traversing array sequentially from the first element till last element and final result is returned as an output.

Data Structures: Introduction

- An Algorithm is a solution of a given problem.
- Algorithm = Solution
- One problem may has many solutions:

e.g.

Searching => to search a given key element in a collection/list of elements.

Searching solutions/algorithms : **linear search & binary search**

Sorting => to arrange data elements in a collection/list of elements either in an ascending order or in a descending order (bydefault in an ascending order).

Sorting solutions/algorithms : **bubble sort, selection sort, insertion sort, quick sort, merge sort etc...**

- If one problem has many solutions we need to select an efficient solution or algorithm.

Data Structures: Introduction

- **Analysis of an algorithm** is a work of determining how much **time** i.e. computer time and **space** i.e. computer memory it needs to run to completion.

- There are two measures of an analysis of an algorithms:

1. Time Complexity of an algorithm is the amount of time i.e. computer time required for it to run to completion.

2. Space Complexity of an algorithm is the amount of space i.e. computer memory required for an algorithm to run to completion.

Asymptotic Analysis: It is a **mathematical** way to calculate time complexity and space complexity of an algorithm **without implementing it in any programming language.**

- In this type of analysis, analysis can be done on the basis of **basic operation** in that algorithm.

e.g. in searching & sorting algorithms comparison is the basic operation and hence analysis gets done on the basis of no. of comparisons, in addition of matrices algorithms addition is the basic operation and hence on the basis of addition operation.

Data Structures: Introduction

"Best case time complexity": if an algo takes min amount of time to run to completion then it is referred as best case time complexity.

"Worst case time complexity": if an algo takes max amount of time to run to completion then it is referred as worst case time complexity.

"Average case time complexity": if an algo takes neither min nor max amount of time to to run to completion then it is referred as an average case time complexity.

"Asympotic Notations":

1. Big Omega (Ω): this notation is used to denote best case time complexity – also called as **asymptotic lower bound**

2. Big Oh (O): this notation is used to denote worst case time complexity - also called as **asymptotic upper bound**

3. Big Theta (θ): this notation is used to denote an average case time complexity - also called as **asymptotic tight bound**

Data Structures: Searching Algorithms

1. Linear Search/Sequential Search:

Step-1: accept key from the user

Step-2: compare the value of key with each array element sequentially by traversing it from the first element till either key is found or maximum till the last element. If key is found then return true otherwise return false.

```
Algorithm LinearSearch( A, size, key){
    for( index = 1 ; index <= size ; index++ ){
    if( key == A[ index ] )
    return true;
    }
    return false;
}
```

Best Case: If key is found at very first position in only 1 no. of comparison then it is considered as a best case and running time of an algorithm in this case is **O(1)** => and hence time complexity = **Ω(1)**

Worst Case: If either key is found at last position or key does not exists, maximum **n** no. of comparisons takes place, it is considered as a worst case and running time of an algorithm in this case is O(n) => and hence time complexity = **O(n)**

Average Case: If key is found at any in between position it is considered as an average case and running time of an algorithm in this case is O(n/2) => and hence time complexity = **θ(n)**

Data Structures: Searching Algorithms

2. Binary Search/Logarithmic Search:

- This algorithm follows **divide-and-conquer** approach.

- To apply binary search on an array prerequisite is that **array elements must be in a sorted manner.**

Step-1 : accept key from the user

Step-2 : calculate **mid position** of an array by the formula, **mid = (left+right)/2** (by means of calculating mid position big size array gets divided logically into two subarrays, from **left to mid-1 = left subarray** & **mid+1 to right = right subarray**).

Step-3 : compare the value of key with element which is at mid position. if key matches with element at mid position means key is found and return true.

Step-4 : if key do not matches then check, is the value of key is less than element which is at mid position, if yes then goto **search key only into the left subarray by skipping whole right subarray otherwise (means of the value of key is greater than element which is at mid position) goto search key only into the right subarray by skipping whole left subarray.**

Step-5 : repeat Step-2, Step-3 & Step-4 till either key is not found or max till the subarray is valid, if subarray becomes invalid means key is not found and hence return false in this case.

Data Structures: Searching Algorithms

```
Algorithm BinarySearch(A, n, key)//A is an array of size "n", and key to be search
{
   left = 1;
   right = n;

   while( left <= right )
   {
     //calculate mid position
     mid = (left+right)/2;
     //compare key with an ele which is at mid position
     if( key == A[ mid ] )//if found return true
       return true;

     //if key is less than mid position element
     if( key < A[ mid ] )
     {
       right = mid-1;//search key only in a left subarray
     }
     else//if key is greater than mid position element
     {
       left = mid+1;//search key only in a right subarray
     }
   }//repeat the above steps either key is not found or max any subarray is valid
   return false;
}
```

Data Structures: Searching Algorithms

Best Case: if the key is found in an array in very first iteration at mid position only **1 no. of comparison** it is considered as a best case and running time of an algorithm in this case is

O(1) => **Ω(1).**

i.e. If key is found at **root position** it is considered as a best case.

Worst Case: if either key is not found or key is found at **leaf position** it is considered as a worst case and running time of an algorithm in this case is O(log n) => **O(log n).**

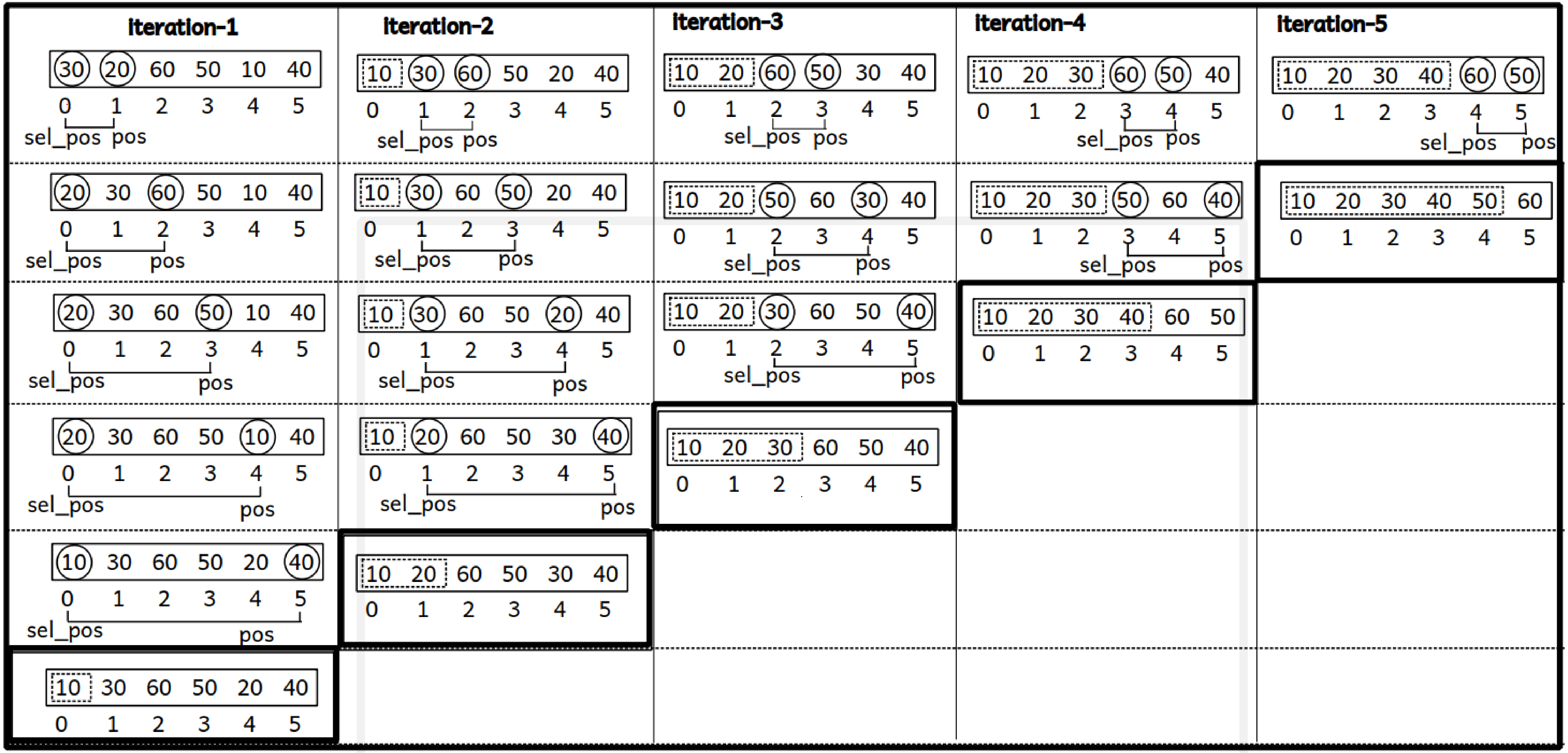
Average Case: if key is found at **non-leaf position** it is considered as an average case and running time of an algorithm in this case is O(log n) => **θ(log n).**

1. Selection Sort:

- In this algorithm, in first iteration, first position gets selected and element which is at selected position gets compared with all its next position elements, **if selected position element found greater than any other position element then swapping takes place** and **in first iteration smallest element gets setteled at first position.**

- In the second iteration, second position gets selected and element which is at selected position gets compared with all its next position elements, again if selected position element found greater than any other position element then swapping takes place and **in second iteration second smallest element gets setteled at second position,** and so on **in maximum (n-1) no. of iterations** all array elements gets arranged in a sorted manner.

Data Structures: Sorting Algorithms



Data Structures: Sorting Algorithms

Best Case : $\Omega(n^2)$

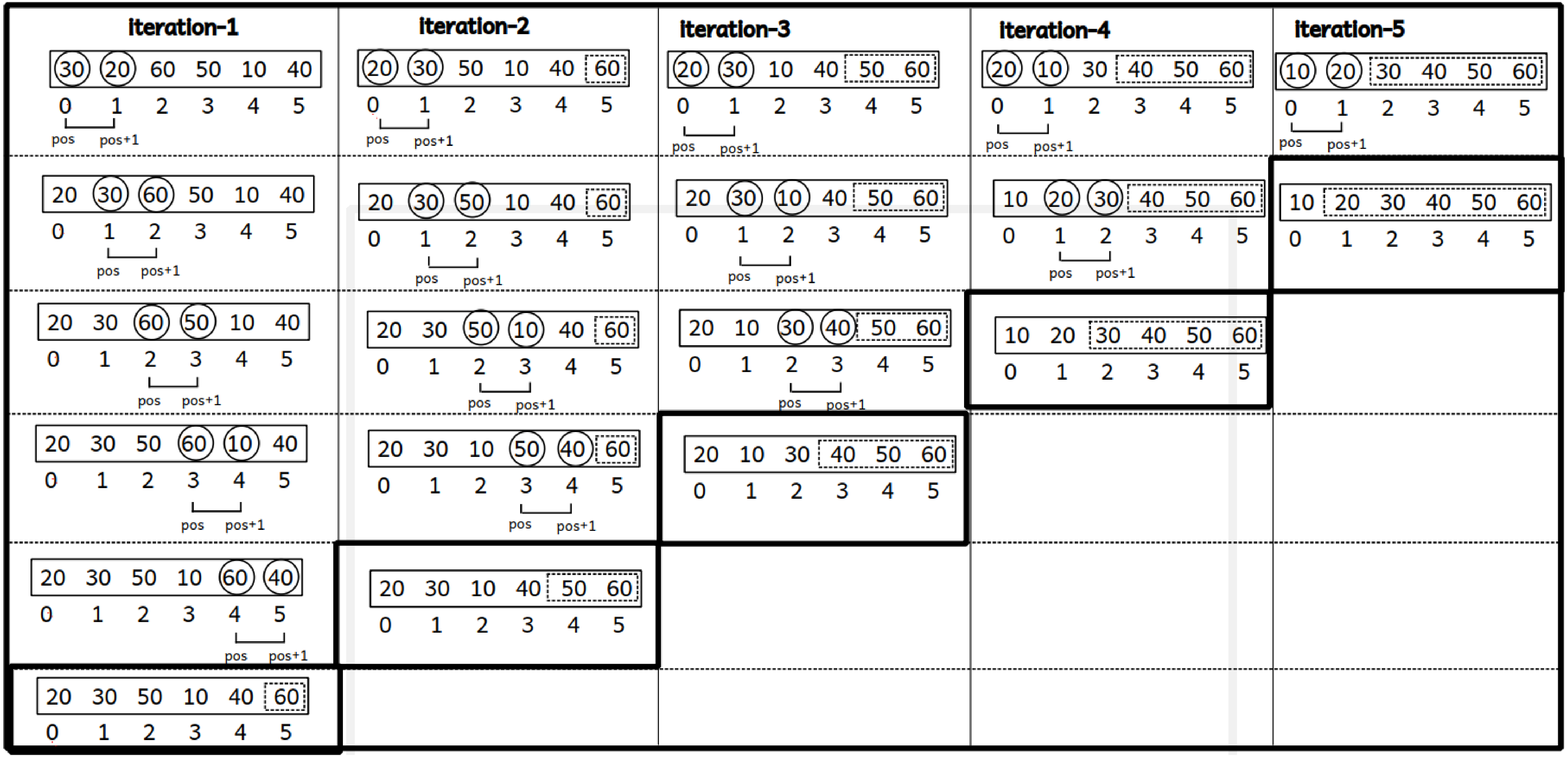
Worst Case : $O(n^2)$

Average Case : $\theta(n^2)$

2. Bubble Sort:

- In this algorithm, **in every iteration elements which are at two consecutive positions gets compared, if they are already in order then no need of swapping between them, but if they are not in order i.e. if previous position element is greater than its next position element then swapping takes place**, and by this logic **in first iteration largest element gets fixed at last position, in second iteration second largest element gets fixed at second last position** and so on, **in max (n-1) no. of iterations** all elements gets arranged in a sorted manner.

Data Structures: Sorting Algorithms



Data Structures: Sorting Algorithms

Best Case : Ω(n) – if array elements are already arranged in a sorted manner.
Worst Case : $O(n^2)$
Average Case : $\theta(n^2)$

3. Insertion Sort:
- In this algorithm, one array is considered logically as two sets in which **initially first set contains one element** and **second set contains remaining all elements i.e. (n-1) no. of elements.**
- in every iteration one element will be picked sequentially from second set referred as a **key element,** which will be inserted into the first set at its appropriate position i.e. while inserting key element into the first set it makes sure that all the elements in a first set are arranged in a sorted manner, and to do this key element gets compared with elements which are there into the first set from **left to right** and wherever its appropriate position found it will be **inserted** into the first set, by repeating this logic in **max (n-1) iterations** all the elements from second set will be inserted into the first set and all array elements gets arranged into the first set in a sorted manner.

Data Structures: Sorting Algorithms

Best Case : Ω(n) – if array elements are already arranged in a sorted manner.

Worst Case : O(n^2)

Average Case: θ(n^2)

- Insertion sort algorithm works efficiently for already sorted input array by design.
- Insertion sort algortihm is an efficient algorithm for smaller input size array.

4. Merge Sort:

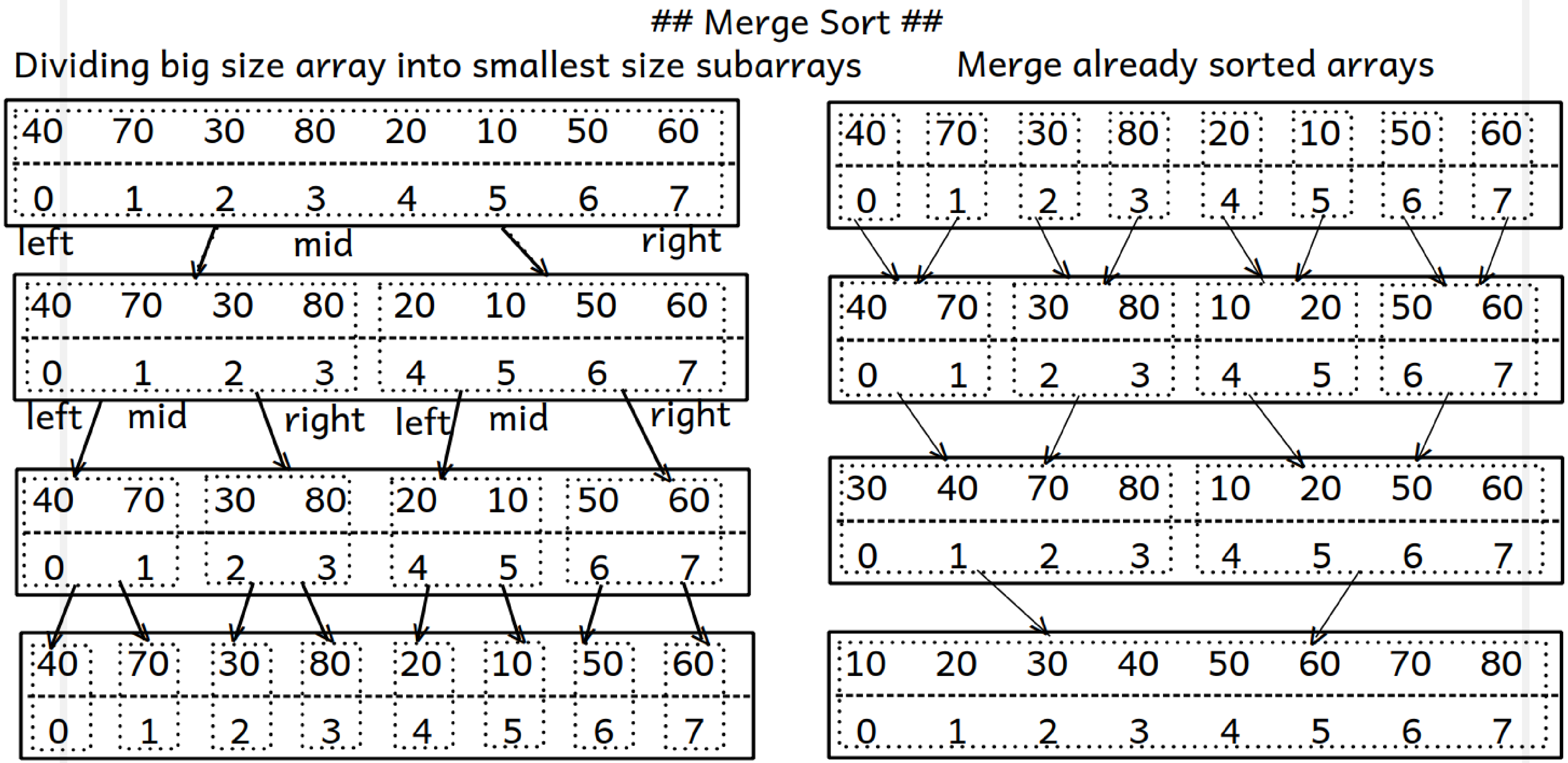
- This algorithm follows **divide-and-conquer** approach.
- In this algorithm:

Step-1: big size array gets divided logically into smallest size subarrays (i.e. having size 1)

Step-2: keep on merge already sorted subarrays into a single array in a sorted manner step by step and finally whole array gets sorted.

- This algorithm works fine for **even** as well **odd** input size array.
- This algorithm takes extra space to sort array elements, and hence its space complexity is more.

Data Structures: Sorting Algorithms



Best Case : $\Omega(n \log n)$
Worst Case : $O(n \log n)$
Average Case : $\theta(n \log n)$

5. Quick Sort:
- This algorithm follows **divide-and-conquer** approach.
- In this algorithm the basic logic is a **partitioning.**
- **Partitioning:**

Step-1: select pivot element (select leftmost element as a pivot element).
Step-2: shift elements towards left side which are smaller than pivot and **shift elements towards right side which are greater than pivot**, due to this, **pivot element gets settled at its appropriate position**, and array gets divided logically into two partitions in such a way that elements which are at left of pivot is referred as **left partition** and elements which are at its right referred as a **right partition.**

Best Case : $\Omega(n \log n)$

Worst Case : $O(n^2)$ – worst case rarely occures

Average Case : $\theta(n \log n)$

- Quick sort algortihm is an efficient sorting algorithm for larger input size array.