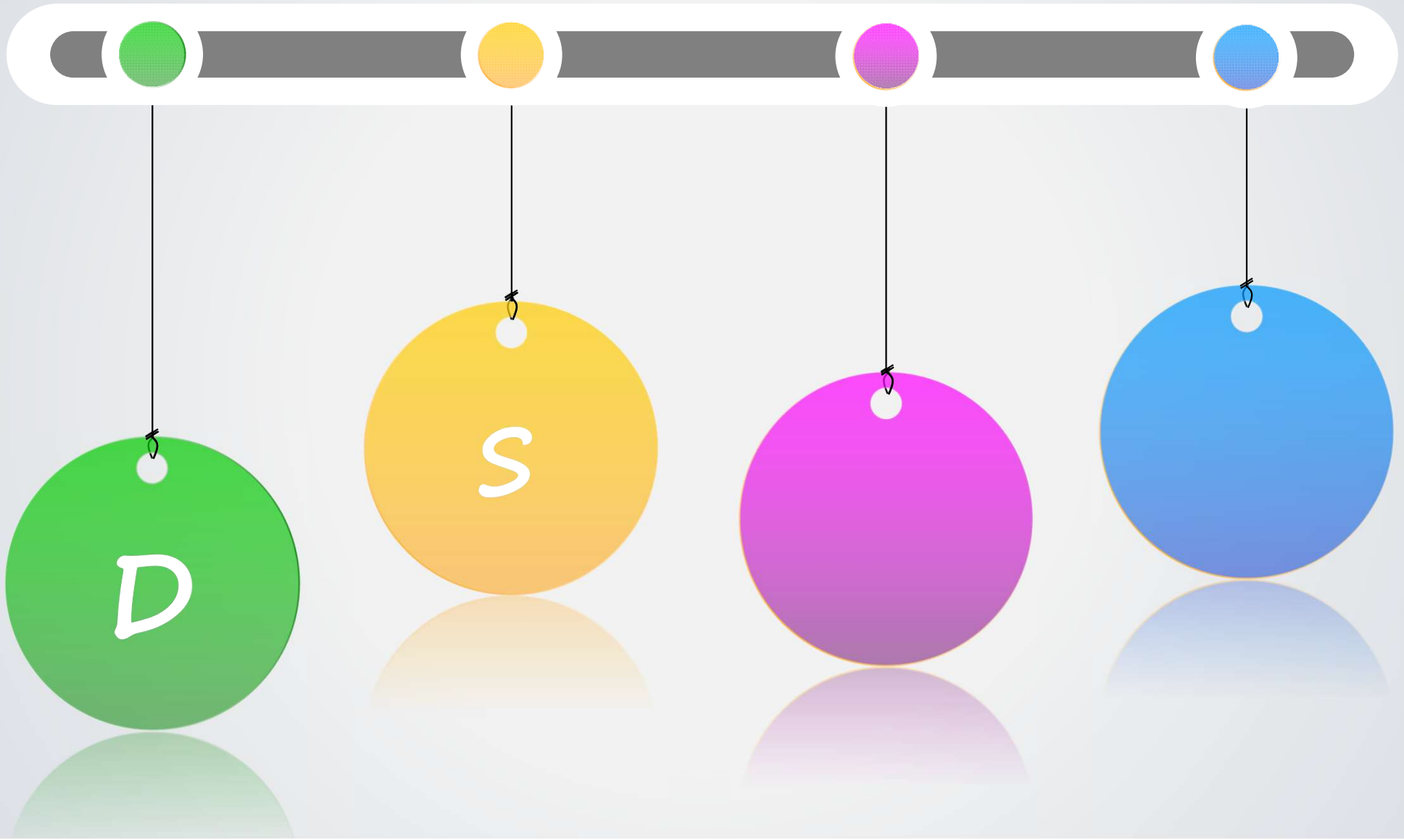
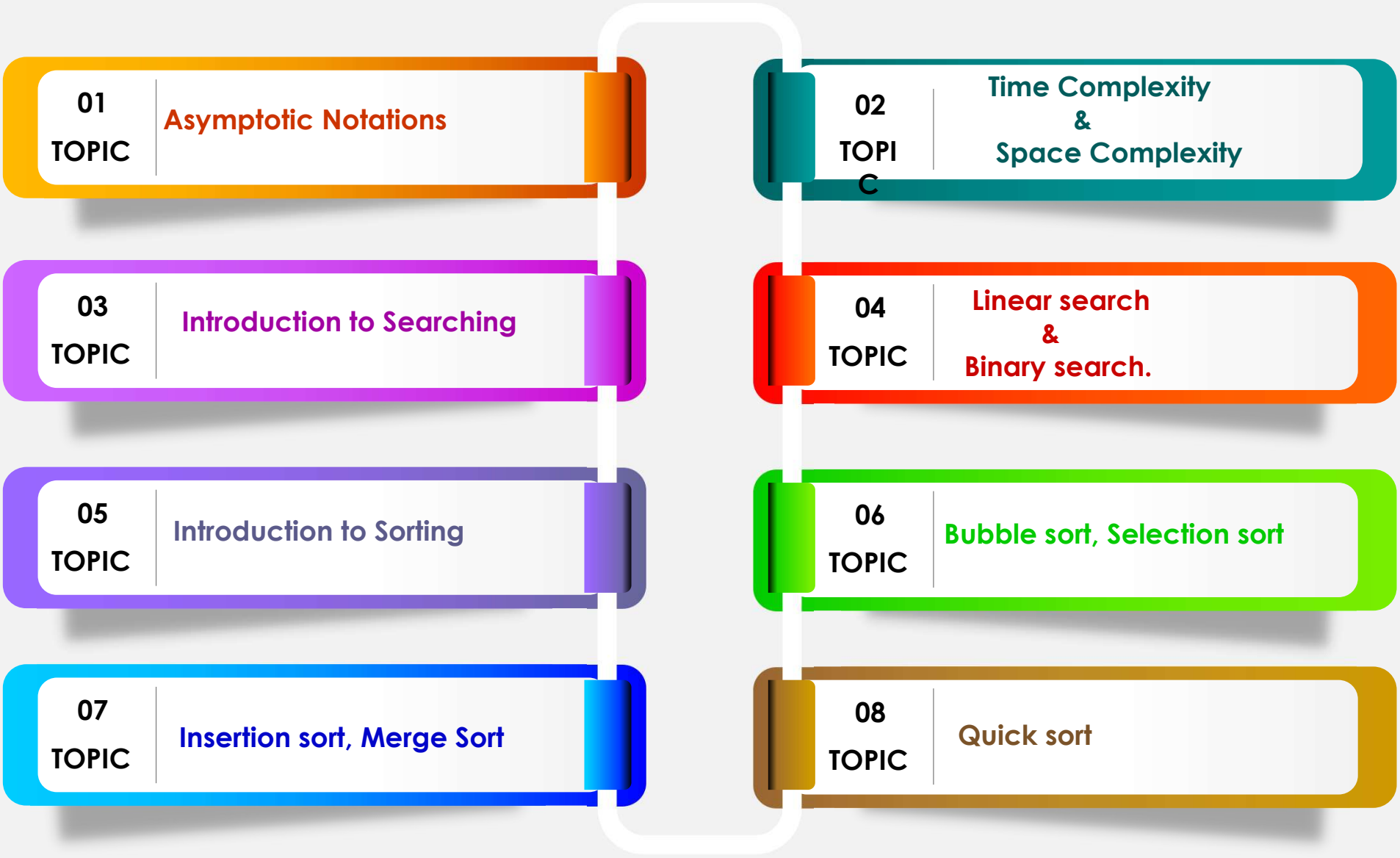

D
S


01
TOPIC
Asymptotic Notations
02
TOPI
C
Time Complexity
&
Space Complexity
03
TOPIC
Introduction to Searching
04
TOPIC
Linear search
&
Binary search.
05
TOPIC
Introduction to Sorting
06
TOPIC
Bubble sort, Selection sort
07
TOPIC
Insertion sort, Merge Sort
08
TOPIC
Quick sort

Course Outcomes (COs)

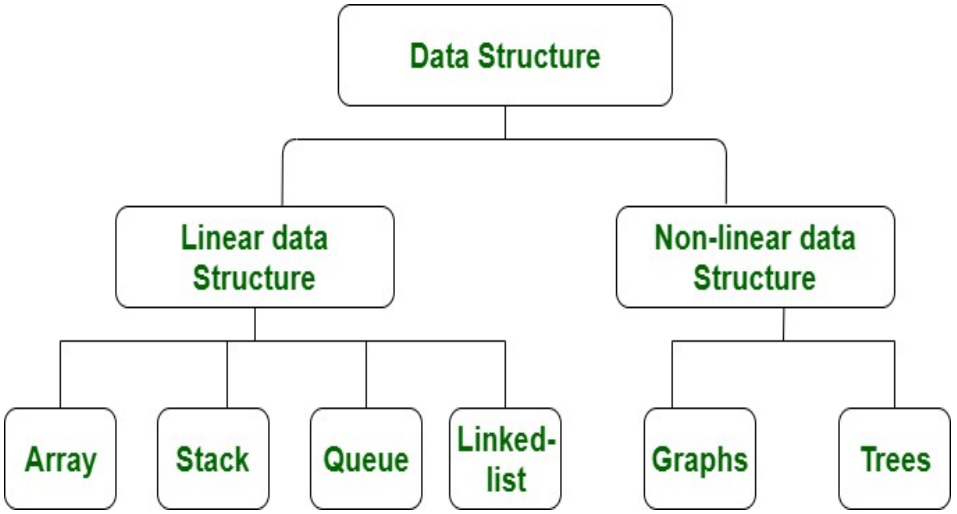
- **Select** appropriate **sorting and searching technique** for a given application.
- Use **various forms of linked lists** to perform **operations** on data efficiently.
- **Build** applications using **stack** data structure for **real time applications.**
- Construct various forms of **Queues** to solve a **real time problem.**
- Make **use of nonlinear data structures** for **organizing data.**

Def of Data Structure

- Data structure is a **particular way of storing and organizing data in a computer**. So that it can be used efficiently.
- Data structure is a specialized format for **organizing and storing data**.
- The **data structure** can be divided into **two basic types**

I. **Linear Data Structures**

II. **Non Linear Data Structures**



Types of Data Structures

i)Linear Data Structures:

- "When **elements** of a data structure are **connected in linear fashion** is known linear data structure".
- Here, the elements are stored in **non-hierarchical** way where **each element has the successors and predecessors** except the first and last element.

Example
Arrays, stacks, queues and **Lists**.

ii)Non Linear Data Structures:

- "When **elements** of a data structure are **connected in Non-linear fashion** is known as Non-linear data structure. Here **data** may be **arranged** in a **hierarchical manner**".

Example
Trees, Graphs, Sets and Tables

i. Types of Linear Data Structures

a) Arrays:

- An array is a **collection of similar type of data items**.

b) Linked List:

- It is a **collection of nodes stored at non-contiguous memory locations**. Each node of the list contains **data** and **pointer**.

c) Stack:

- Stack is **a linear list** in which **insertion and deletions are allowed only at one end.**

d) Queue:

- Queue is a **linear list** in which elements can be **inserted only at one end** called **rear** and **deleted only at the other end** called **front.**

ii.Types of Non-Linear Data Structures

a) Trees:

- ✓ Trees are **multilevel data structures** with a **hierarchical relationship** among its elements known as nodes.
- ✓ Tree **data structure is based on the parent-child relationship** among the nodes.

b) Graphs:

- ✓ Graphs can be **defined as the pictorial representation** of the **set of elements** (represented by vertices) **connected** by the links known **as edges.**
- ✓ A **graph** is **different from tree** in the sense that a **graph can have cycle** while the **tree can not have the cycle.**

Operations on Data Structures

- **Insertion:** Insertion means **addition of a new data element** in a data structure.
- **Deletion:** Deletion means **removal of a data element** from a data structure if it is found.
- **Searching:** It is used to **find out the location of the data item** in a data structures if it is found.
- **Traversing:** It is used to **access each data item exactly once** so that it can be processed.
- **Sorting:** It is used to **arrange** the **data items** either in **ascending** or **descending order**.
- **Merging:** It is used to combine the **data items of two sorted files** into single file in the sorted form.

Algorithm

- "An algorithm is the **process of solving the problem step by step**".
- **"**An algorithm is the **step-by-step method of performing any task"**.
- All algorithms must **satisfy** the **following criteria**:

i	**Input**	: Ever algorithm should **take at least one input.**
ii	**Output**	: Every algorithm must **produce at least one output.**
iii	**Definiteness**	: **Each instruction** is **clear** and **with out any confusion.**
iv	**Finiteness**	: The algorithm should be **terminates after a finite number of steps**.
v	**Effectiveness**	: **Every instruction** must be **easily converted** into **program**.
vi	**Language independent**	: An algorithm must be **language-independent.**
vii	**Efficient**	: Algorithm must **take less time** and **less space** for its completion.

Algorithm Analysis

- For **any problem**, there are **N number of solution**. This is true in general.

Example:

- ✓*If **I have a problem** and I **discuss** about the problem with all of **my friends**, they will all **suggest me different solutions.***
- ✓ ***I am** the one who has to **decide which solution** is the **best** based on the circumstances.*

- Similarly for **any problem** which must be **solved using an algorithm**, there can be **number of solutions.**
- The **efficiency of an algorithm** can be decided by **measuring** the **performance** of an algorithm.
- We can **measure the performance** of an algorithm by computing **amount of time and storage requirement**
- We have to find **best solution** based on **time complexity** and **space complexity.**

What is Time Complexity?

Def. of Time complexity:

-Time complexity is defined as the **total amount of time required to run the algorithm**.

- The time complexity of algorithms is most commonly expressed using the **big O notation**.
- We can calculating the Time Complexity by using following methods:
  i. **Input size**
  ii. **Frequency Count**

i. Input size

- The input size is defined as **the number of inputs required by each statement.**

  Step 1: First, we will compute **input size of each statement**.

  Step 2: **Calculate** the **total** as sum of all inputs .

  Step 3: **Consider** the time complexity as **Higher degree polynomial** and neglect all other constants and other parameters .

Example:

1	Algorithm add (A, B, C, n)	0
2	{	0
3	for i:=1 to n do	n + 1
4	for j:=1 to n do	n (n + 1)
5	C[i,j] := A[i,j] + B[i,j];	n*n
6	}	0
	Total No of Inputs:	$2n^2 + 2n + 1$

1	Algorithm add (A, B, C, m, n)	0
2	{	0
3	for i:=1 to 5 do	5 + 1
4	for j:=1 to 5 do	5 (5 + 1)
5	C[i,j] := A[i,j] + B[i,j];	5*5
6	}	0
	Total No of Inputs:	61

- So, the Time complexity of the above algorithm is: $O(n^2)$

ii. Frequency count

- The Frequency count is defined as **total number of times that each statement run in the machine.**

  Step 1: First, we will compute **no of times that each statement will run**.

  Step 2: **Calculate the frequency count.**

  Step 3: **Consider** the time complexity as **Higher degree polynomial** and neglect all other constants and other parameters .

Example:

1	**Algorithm add (A, B, C, n)**	**1**
2	**{**	**1**
3	**for i:=1 to n do**	**n + 1**
4	**for j:=1 to n do**	**n (n + 1)**
5	**C[i,j] := A[i,j] + B[i,j];**	**n*n**
6	**}**	**1**
Total frequency count:		**$2n^2 + 2n + 4$**

1	Algorithm add (A, B, C, m, n)	1
2	{	1
3	for i:=1 to 5 do	5 + 1
4	for j:=1 to 5 do	5 (5 + 1)
5	C[i,j] := A[i,j] + B[i,j];	5*5
6	}	1
Total frequency count:		**64**

- So, the Time complexity of the above algorithm is: **$O(n^2)$**

Space Complexity

- The space complexity can be defined as **amount of memory required by an algorithm** to run.
- To compute the space complexity we use two factors: **constant** and **instance characteristics**. The space requirement **S(p)** can be given as: **S(p) = C + Sp.**
- Where **C** is a constant i.e**. fixed part** and it denotes the **space of inputs and outputs**. This space is an amount of space taken by instruction, variables and identifiers.
- **Sp** is a space **dependent upon instance characteristics**. This is a variable part whose space requirement depends on particular problem instance.

Asymptotic Notations

- Asymptotic notation is a mathematical way of representing time complexity.
  - Using asymptotic notations we can give time complexity as **"fastest possible", "slowest possible"** or **"average time".**
  - Various notations such as
    - ✓**Omega Notation (Ω)**
    - ✓**Theta Notation(θ)**
    - ✓**Big Oh Notation (O)**

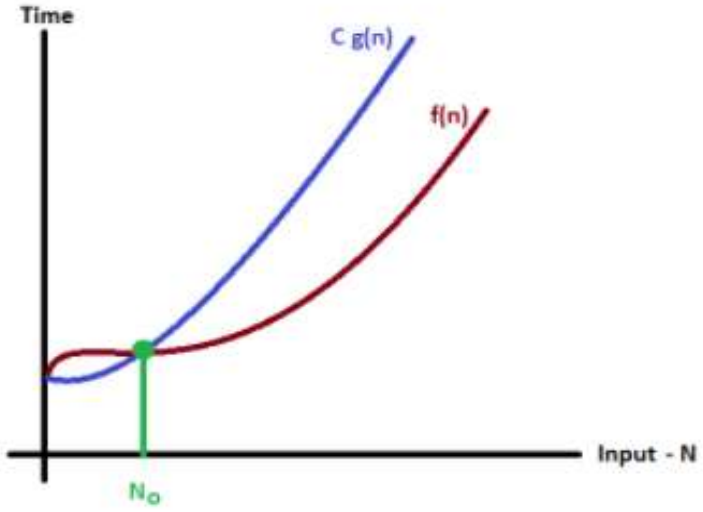
i) Big Oh Notation

- Big Oh notation **denoted by 'O'** is a method of representing the **upper bound of algorithm's running time.**

- Using big oh notation we can give **longest amount of time taken by the algorithm** to complete.

- If f(n) and g(n) are the two functions defined for positive integers then it is represented as:

$$f(n) \leq c.g(n) \quad \text{for all} \quad n \geq No$$



ii) Omega Notation

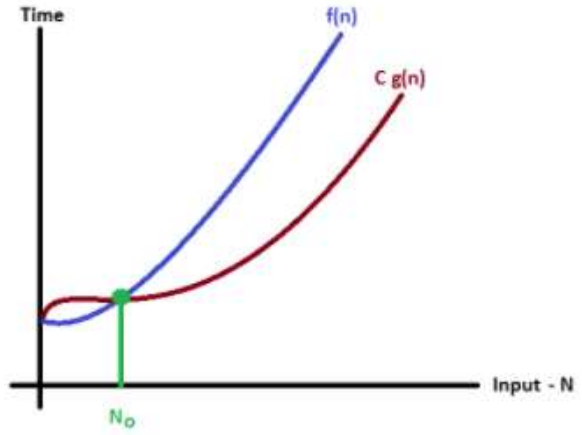
•Omega notation **denoted as 'Ω'** is a method of representing the **lower bound of algorithm's running time.**

•Using omega notation we can denote **shortest amount of time taken by algorithm** to complete.

•It determines what is the **fastest time that an algorithm can run**.

•If f(n) and g(n) are the two functions defined for positive integers,then f(n) = Ω (g(n))

$f(n) >= c.g(n)$ for all $n \geq No$ and $c > 0$



iii) Theta Notation

•Theta notation denoted as ‘θ’ is a method of **representing running time between upper bound and lower bound.**

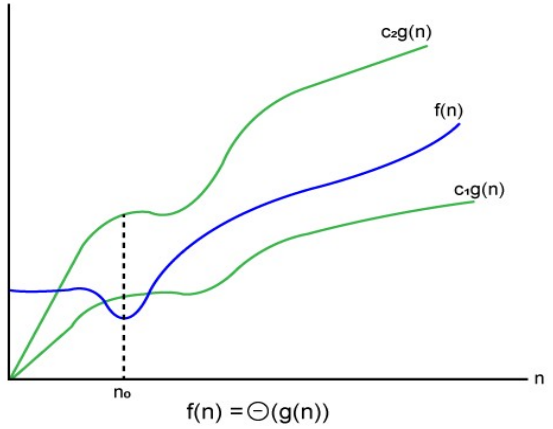
•The theta notation mainly describes the **average case scenarios.**

•Every time, an algorithm does not perform worst or best, in real-world problems, algorithms mainly fluctuate between the worst-case and best-case.

•Let f(n) and g(n) be the functions of n where n is the steps required to execute the program then:

•where the function is bounded by two limits, i.e., upper and lower limit, and f(n) comes in between.

c1.g(n)<=f(n)<=c2.g(n)



Best Case, Worst Case and Average Case Analysis

i)Best case time complexity:

✓If an algorithm takes **minimum amount of time** to run to completion for a specific set of input then it is called **best case time complexity**.

✓EX:**While searching** a particular elements by using sequential search **we get the desired element at first place itself** then it is called best case time complexity.

ii)Worst case time complexity:

✓If an algorithm takes **maximum amount of time** to run to **completion for a specific** set of input then it is called **worst case time complexity**.

✓**EX:While searching** an element by using linear searching method if **desired element is placed at the end of the list** then we get worst time complexity.

Best Case, Worst Case and Average Case Analysis

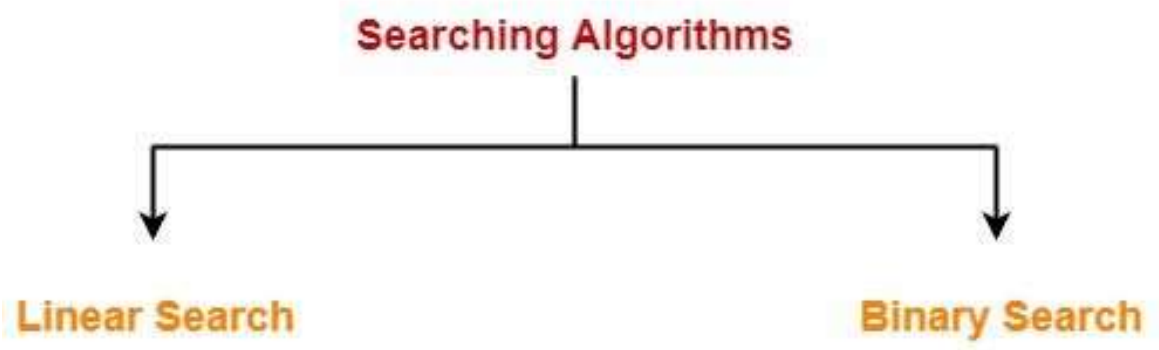
iii)Average case time complexity:

✓The time complexity that we get **for certain set of inputs** is as a average **same, such** time complexity is called **average case time complexity**.

Searching

- Searching is a **process of finding a particular element** among list of elements.
- The search is **successful** if the required **element is found**.
- Otherwise, the search is **unsuccessful.**
- searching of an element in the given array may be carried out in the following two ways

i. **Linear Search**
ii. **Binary Search**



Linear Search

- Linear Search is the simplest searching algorithm.
- It searches for an element **by comparing** it with **each element of the array one by one.**

  So, it is also called as **Sequential Search.**
- Linear Search Algorithm is applied when
  i. The given **array is unsorted** or the elements are unordered.
  ii. The list of **data items is smaller.**

How to Apply Linear Search

Step 1 : Start

Step 2: Read **the list of element** from the user.

Step 3: Read the **search element** from the user.

Step 4: **Compare** the **search element** with the **first element in the list.**

Step 5: If **both are matched**, then display "**Given element is found!!!**" and go to step 9

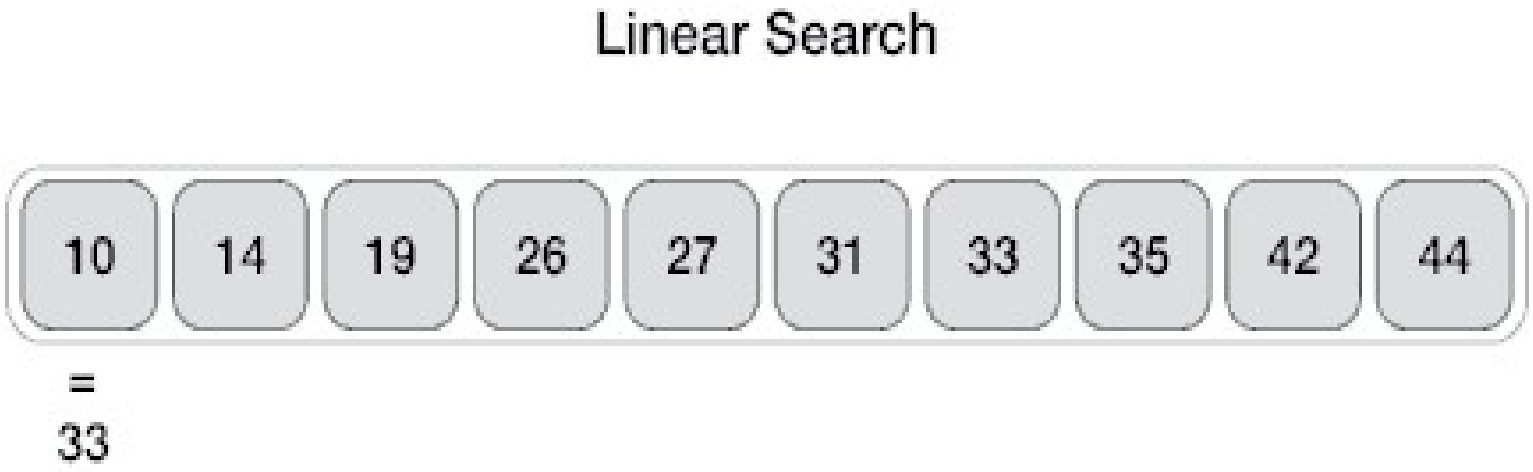
Step 6: If **both are not matched**, then **compare search element with the next element** in the list.

Step 7: **Repeat steps 5 and 6until** search element is compared with last element in the list.

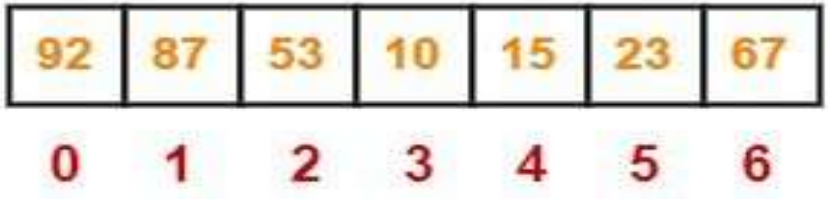
Step 8: If **last element** in the list also **doesn't match**, then display "**Element is not found!!!" .**

Step 9 : Stop

Linear Search

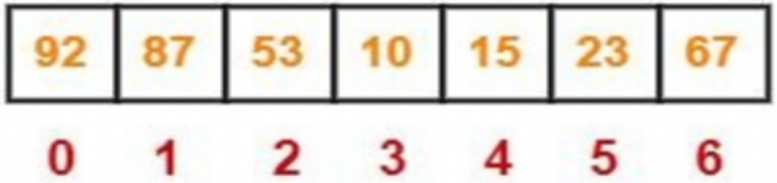


Linear Search Example



- Consider,**Element 15** has to be **searched** in it using Linear Search Algorithm.
- Linear Search algorithm **compares** element 15 with **all the elements of the array one by one**.
- It **continues searching** until **either** the element **15 is found** or **all the elements are searched**.

Linear Search Example



Step-01:

- ✓ It compares element 15 with the 1 st element 92.
- ✓ Since 15 ≠ 92, so required element is not found.
- ✓ So, it moves to the next element.

Step-02:

- ✓ It compares element 15 with the 2 nd element 87.
- ✓ Since 15 ≠ 87, so required element is not found.
- ✓ So, it moves to the next element.

Step-03:

- ✓ It compares element 15 with the 3 rd element 53.
- ✓ Since 15 ≠ 53, so required element is not found.
- ✓ So, it moves to the next element.

Step-04:

- ✓ It compares element 15 with the 4 th element 10.
- ✓ Since 15 ≠ 10, so required element is not found.
- ✓ So, it moves to the next element.

Step-05:

- ✓ It compares element 15 with the 5 th element 15.
- ✓ Since 15 = 15, so required element is found.
- ✓ Now, it stops the comparison and returns index 4 at which element 15 is present.

Linear Search Program

```
#include"stdio.h"
void linear(int [],int,int);
int main()
{
        int sai[10],n,key,i;
        printf("Enter n value:");
        scanf("%i",&n);
        for(i=0;i<n;i++)
                scanf("%i",&sai[i]);
        printf("Enter the search element:");
        scanf("%i",&key);
        linear(sai,n,key);
}
```

```
void linear(int sai[],int n,int key)
{
        int pos=-1;
        for(i=0;i<n;i++)
        {
                if(sai[i]==key)
                    pos=i+1;
        }
        if(pos==-1)
        printf("Key is not present in the array");
        else
        printf("Key is  present at %i  position",pos);
}
```

Linear Search time complexity analysis

Best case:

- ✓In the best case, The **element being searched** may be **found at the first position**.
- ✓In this case, the search terminates in success with just one comparison.
- ✓Thus in best case, linear search algorithm **takes O(1) operations**.

Worst Case:

- ✓In the worst case, The **element being searched** may be **present at the last position** or **not present in the array at all**.
- ✓In this case, the search terminates in failure **with n comparisons**.
- ✓Thus in **worst case**, linear search algorithm **takes O(n) operations.**

Time Complexity of Linear Search Algorithm is O(n)

Here, n is the number of elements in the linear array.

Binary Search

- Binary Search is one of the **fastest searching algorithms.**
- It works on the principle of **divide and conquer technique.**
- Binary Search Algorithm can be applied **only on Sorted arrays.**
- So, the **elements must be arranged** in-Either **ascending order** if the elements are numbers, Or **dictionary order** if the elements are strings.
- To apply binary search on an **unsorted array**, **First, sort** the array using some sorting technique. **Then, use binary search algorithm.**

How To Apply Binary Search

i. Consider-There is a linear array ‘a’ of size ‘n’.

ii. Binary search algorithm is being used to **search an element “key”**in an array.

iii. Variables **low and high** keeps track of **the index of the first and last element** of the array or sub array in which the element is being searched at that instant.

iv. Calculate the **Mid=(low+high)/2**

v. Variable **mid keeps track of the index of the middle element** of that array or sub array in which the element is being searched.

vi. If search ends in success, **it sets POS to the index of the element otherwise it sets POS to -1**.

How To Apply Binary Search

▪ Binary Search Algorithm searches an element by comparing it with the middle most element of the array. Then, following three cases are possible

i.Case-01:

✓If the element being searched is **found** to be **the middle** most element**, its index is returned**.

ii.Case-02:

✓If the element being searched is **found** to be **greater than the middle** most element,then its search is further **continued in the right sub array** of the middle most element.

iii. Case-03:

✓If the element being searched is **found** to be **smaller than the middle most** element,then its search is further **continued in the left sub array** of the middle most element.

▪ This process keeps on repeating on the sub arrays until the desired element is found or size of the sub array reduces to zero.

Binary Search Algorithm

Step 1:Start

Step 2: SET low= 0 high = n-1 status = - 1

Step 3: while(**low <=high**)

mid = (low + high)/2

if (a[mid] == KEY)

status = mid and Go to Step 4

else if (a[mid] > KEY)

high = mid - 1

else

low = mid + 1

End while

Step 4: if (status == -1)

Write "VALUE IS NOT PRESENT IN THE LIST“

else

Write "VALUE IS PRESENT IN THE LIST“

Step 5: Stop

Binary Search Example

- Consider,We have a following sorted array.
- Element 15(key) has to be searched in it using Binary Search Algorithm.

3	10	15	20	35	40	60
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

Step-01:

To begin with, we take **low=0 and high=6,key=15**

While(low<=high)

{

We compute location of the middle as:

mid= (low + high) / 2

mid= (0 + 6) / 2

mid= 3

Here,

i) a[mid] ==key

a[3] ==15

20 == 15 condition false .

So, we have to check next condition

ii) a[mid]>key

20>15

The condition is True so we have to update the high

high=mid-1

=3-1

=2

}

Step-02:low remains unchanged, **so low=0,high=2,key=15**

While(low<=high)

{

We compute location of the middle as:

mid= (low + high) / 2

mid= (0 + 2) / 2

mid= 1

Here,

i) a[mid] ==key

a[1] ==15

10 == 15 condition false .

So, we have to check next condition

ii) a[mid]>key

10>15

The condition is false

so we have need not to update the high

so we have to update the low

low=mid+1

=1+1 =>2

}

Step-03:

high remains unchanged, so high=2,low=2,key=15

While(low<=high)

{

We compute location of the middle as:

mid= (low + high) / 2

= (2 + 2) / 2

= 2

Here,

i) a[mid] ==key

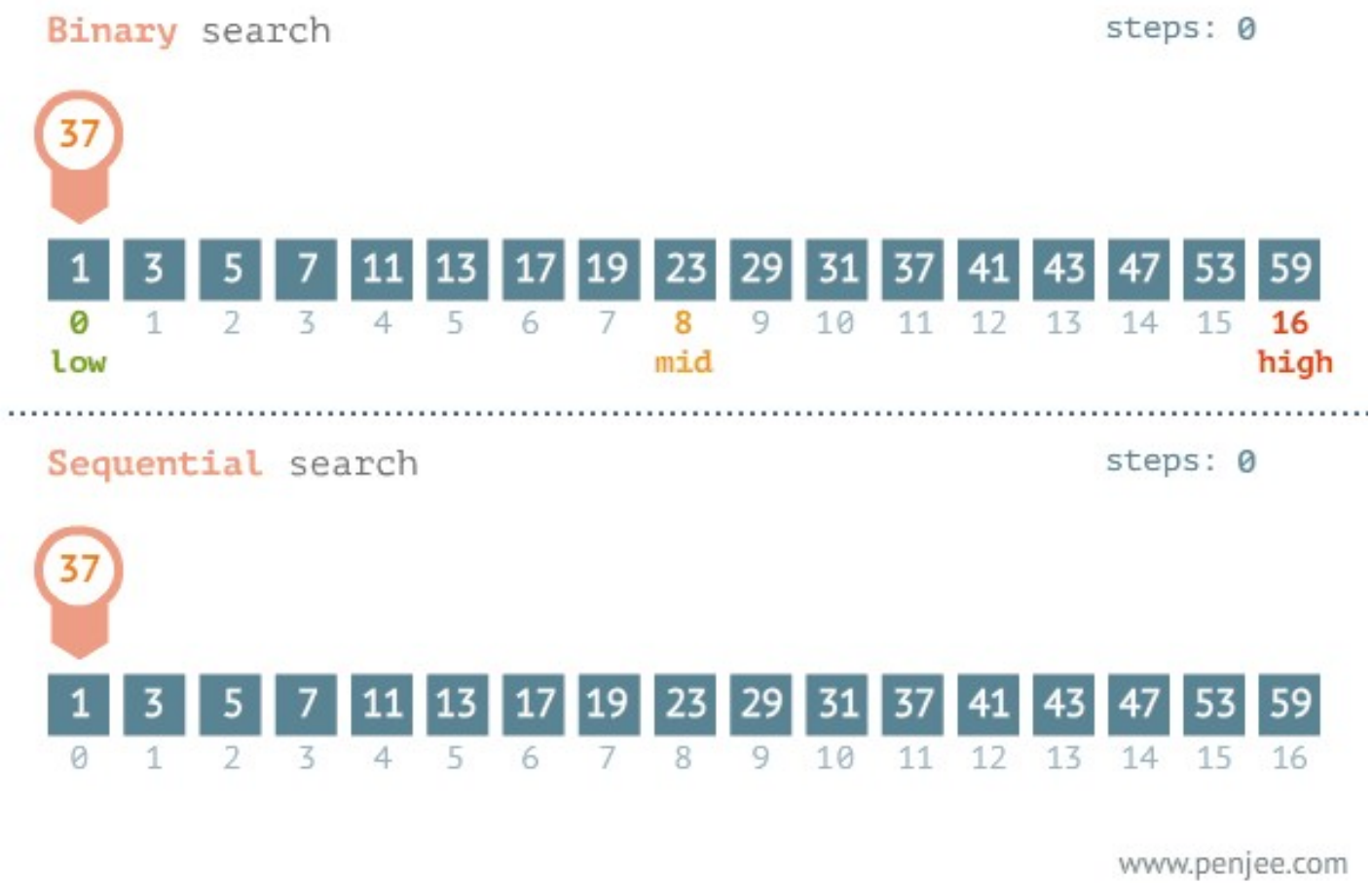
a[2] == 15

15==15

which matches to the element being searched.

So, our search terminates in success and index 2 is returned.

}


Binary search
steps: 0
37
1 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
low
mid
high
Sequential search
steps: 0
37
1 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
www.penjee.com

Binary Search for 50 in 7 elements Array

Binary Search without recursion

```
#include <stdio.h>
int binarySearch(int [],  int , int ,int );
int main()
{
        int a[20],n, status, key,i;
        Printf("Enter List size:");
        Scanf("%i",&n);
        for(i=0;i<n;i++)
              Scanf("%i",&a[i]);
        Printf("Enter the key value:");
        Scanf("%i",&key);
        status = binarySearch(a,  0, n - 1,key);
        if (status== -1)
              printf("Key not found");
        else
        printf("Key  found at index %i", status);
}
```

```
int binarySearch(int a[],  int low,  int high, int key)
 {
        while (low <= high)
        {
              int mid = (low + high ) / 2;
               if (a[mid] == key)
                        return mid;
               if (a[mid] > key)
                        high = mid - 1;
              else
                        low = mid + 1;
        }
return   -1;
 }
```

Binary Search time complexity analysis

Best case:

- ✓In the best case, The **element being searched** may be **found at the mid position**.
- ✓In this case, the search terminates in success with just one comparison.
- ✓Thus in best case, linear search algorithm **takes O(1) operations**.

Binary Search time complexity analysis

Worst case:

- Binary Search terminates after k iterations.
- At each iteration, the array is divided by half. So let's say the length of array at any iteration is n.

i. At **Iteration 1**,Length of array = **n**

ii. At **Iteration 2**,Length of array = $n/2$

iii. At **Iteration 3**,Length of array = $(n/2)/2 = n/2^2$

iv. Therefore, after **Iteration k**,

Length of array = $n/2^k$

v. Also, we know that afterAfter k divisions, the **length of array becomes 1**

vi. Therefore Length of array = $n/2^k = 1$ => $n = 2^k$

vii. Applying log function on both sides:

=> $\log_2(n) = \log_2(2^k)$

=> $\log_2(n) = k \log_2(2)$ (As $\log_a(a) = 1$)

Therefore,

=> $k = \log_2(n)$

Hence, the time complexity of Binary Search is

$\log_2(n)$

Bubble Sort

- Bubble sort will **start** by comparing the **first element** of the array with the **second element**, if the **first element is greater than** the **second element**, it will **swap** both the elements, and then **move on to compare the second and the third element**, and so on.

- It is a comparison-based algorithm in which **each pair of adjacent elements is compared** and the **elements are swapped if they are not in order.**

- In **bubble sort**, after **completion of each iteration the largest element** in the list **reached** to its **original place.**


5 1 12 -5 16
unsorted


5 1 12 -5 16
5 > 1, swap
1 5 12 -5 16
5 < 12, ok
1 5 12 -5 16
12 > -5, swap
1 5 -5 12 16
12 < 16, ok


1 5 -5 12 16
1 < 5, ok
1 5 -5 12 16
5 > -5, swap
1 -5 5 12 16
5 < 12, ok


1 -5 5 12 16
1 > -5, swap
-5 1 5 12 16
1 < 5, ok


-5 1 5 12 16
-5 < 1, ok


-5 1 5 12 16
sorted

Bubble Sort Algorithm

The steps involved in bubble sort(for sorting a given array in ascending order):

Step 1: Starting with the **first element(index = 0), compare the current element with the next element** of the array.

Step 2: If the **current element is greater than the next element** of the array, **swap them.**

Step 3: If the **current element is less than the next element**, **move to the next element. Repeat Step 1 and 2.**

```
BubbleSort( list )
begin
        for i = 0 to n-1
                for j = 0 to n-i-1
                        if( list[j] > list[j+1])then
                        swap( list[j], list[j+1] )
                        end if
                end for
        end for
end
```

Bubble Sort Example

- See the animation below for more understanding:

Bubble Sort Program

```
#include <stdio.h>
int main()
 {
        int  a[100], i, n;
        printf("Enter the list Size ");
        scanf("%d", &n);
        printf("Enter %d elements : ", n);
        for(i = 0; i < n; i++)
                scanf("%d", &a[i]);
        bubbleSort(a, n);
        printf("\n Elements after Sorting: ");
        for(i = 0; i < n; i++)
                printf("%d ", a[i]);
 }
```

```
void bubbleSort(int list[], int n)
{
        int i, j, temp;
        for(i = 0; i < n; i++)
        {
         int status = 0;
                for(j = 0; j < n-i-1; j++)
                {
                        if( list[j] > list[j+1])
                        {
                                temp = list[j];
                                list[j]= list[j+1];
                                list[j+1]= temp;
                                status=1;
                        }
                 if(status==0)
                break;

                }
        }
}
```

Bubble Sort Worst case Time complexity

Number of comparisons will be done in the 1st pass (1st Iteration) **: n-1**

Number of comparisons will be done in the 2nd pass (2nd Iteration) **: n-2**

Number of comparisons will be done in the 3rd pass (3rd Iteration) **: n-3**

……

……

Number of comparisons will be done in the $(n-2)^{th}$ pass **: 2**

Number of comparisons will be done in the $(n-1)^{th}$ pass **: 1**

Total Time = (n-1) + (n-2) + (n-3) + ….. + 3 + 2 + 1

= n(n-1)/2

=O(n^2)

Hence the worst case **time complexity** of Bubble Sort is **O(n^2)**.

Bubble Sort Time complexity

- **Best case time complexity** :Here, we need not to go for second iteration because at the end of the first iteration we will exit from the both the loops, so only one iteration completed, **the no of comparisons in first iteration is (n-1)** after ignoring all the constants the time complexity is **Ω(n)** .
- The **space complexity** for Bubble Sort is **O(1)**, because only a single additional memory space is required i.e. for ***temp*** variable.
- Following are the Time and Space complexity for the Bubble Sort algorithm.

i. Worst Case Time Complexity [Big-O] : **$O(n^2)$**

ii. Best Case Time Complexity [Omega] : **Ω(n)**

iii. Average Time Complexity [Theta] : **Θ (n^2)**

iv. Space Complexity : **O(1)**

Selection Sort

- This algorithm **divides** the **given input list** into **two parts**, first part which is already **sorted** and second part which is still **not sorted**.
- Initially the **sorted list is empty** and **unsorted list** is contains **all the items**.
- We will **starts** with **finding** the **smallest element** in unsorted list and **swaps it with the leftmost element in the unsorted list.**
- The **smallest element** which was **swapped is part** of **sorted list** now so **increase** the **starting index of unsorted list for the next iteration.**
- **Repeat the process till the entire list is sorted** (sorted list have all the element and unsorted list is empty).
- In **every iteration of selection sort, the minimum element** (considering ascending order) from the unsorted list is **picked and moved to the sorted list.**

Selection Sort Algorithm

Step 1 :Start

Step 2 : Set MIN to location 0

Step 3 : Search the minimum element in the list

Step 4 : Swap MIN with the leftmost element

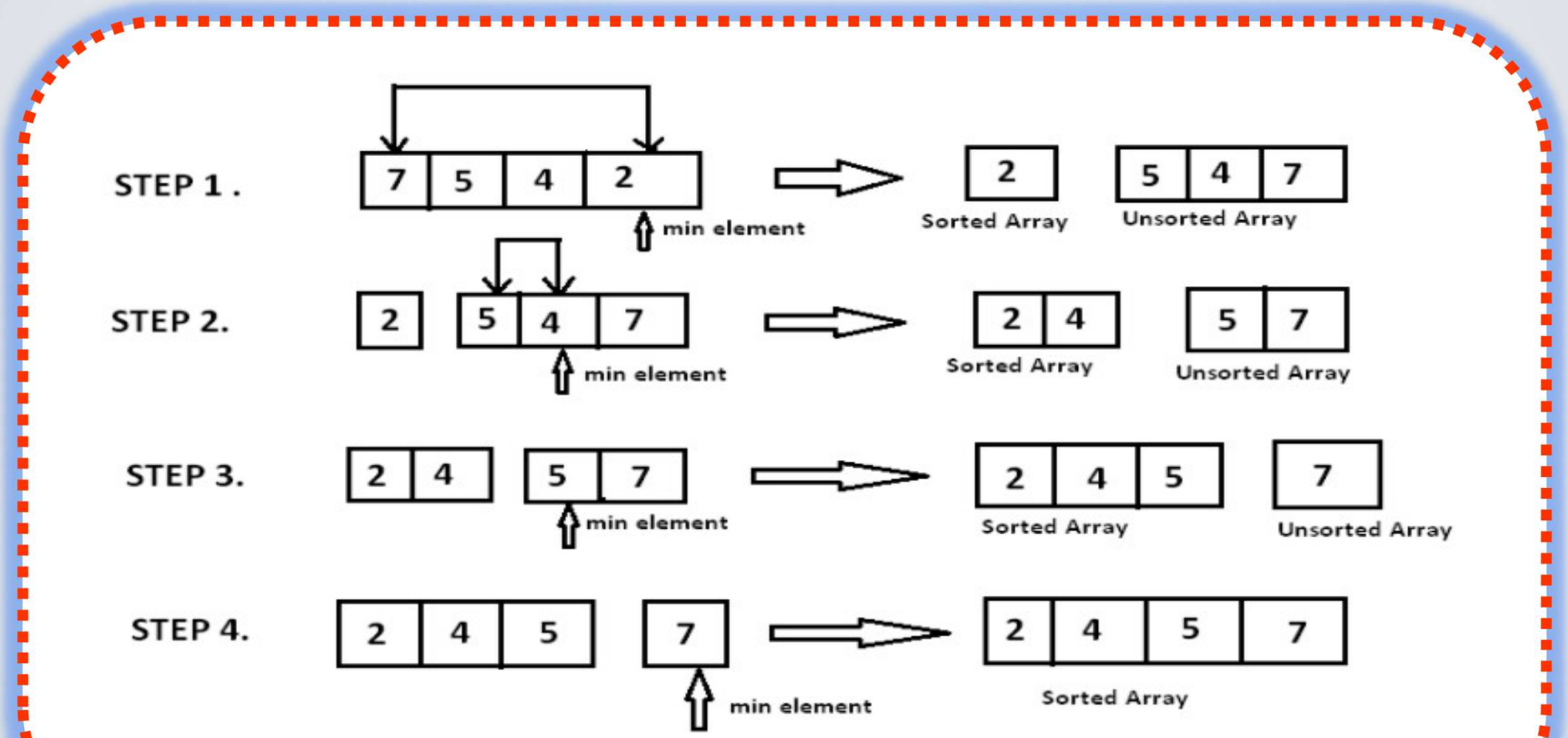
Step 5 : Increment MIN to point to next element

Step 6 : Repeat until list is sorted

Step 7 : Stop

```
Selection Sort (List, n)
Begin
        for i= 0 to n-1
                Set Min =i
                for  j=i+1  to  n
                        if (List [j] < List [Min])
                                Set Min =j
                        end if
                end for
        if(min!=i)
                swap( List[ i], List[Min] )
        end for
End
```

Selection Sort Example



Selection Sort

✓See the animation below for more understanding.

Selection Sort Program

```
#include <stdio.h>
int main()
{
        int  a[20], i, n;
        printf("Enter the list Size ");
        scanf("%d", &n);
        printf("Enter %d elements : ", n);
        for(i = 0; i < n; i++)
                scanf("%d", &a[i]);
        selectionsort(a, n);
        printf("\n Elements after Sorting: ");
        for(i = 0; i < n; i++)
                printf("%d ", a[i]);
}
```

```
void selectionsort (int a[ ], int n)
{
        int min,i,j;
        for( i = 0; i < n-1 ; i++)
        {
                min = i ;
                for(j = i+1; j < n ; j++ )
                {
                        if(a[ j ] < a[ min])
                        {
                                min = j ;
                        }
                }
                if(min!= i)
                {
                        temp=a[i];
                        a[i]=a[min];
                        a[min]=temp;
                }
        }
}
```

Selection Sort Worst case Time complexity

Number of comparisons will be done in the 1st pass (1st Iteration) **: n-1**

Number of comparisons will be done in the 2nd pass (2nd Iteration) **: n-2**

Number of comparisons will be done in the 3rd pass (3rd Iteration) **: n-3**

……

……

Number of comparisons will be done in the $(n-2)^{th}$ pass **: 2**

Number of comparisons will be done in the $(n-1)^{th}$ pass **: 1**

Total Time = (n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1

= n(n-1)/2

=O(n^2)

Hence the worst case **time complexity** of Bubble Sort is **O(n^2)**.

Selection Sort Time complexity

- **best case time complexity** will be **$O(n^2)$**, it is when the list is already sorted.
- The **space complexity** for selection Sort is **O(1)**, because only a single additional memory space is required i.e. for ***temp*** variable.
- Following are the Time and Space complexity for the selection Sort algorithm.

i. Worst Case Time Complexity [Big-O] : **$O(n^2)$**

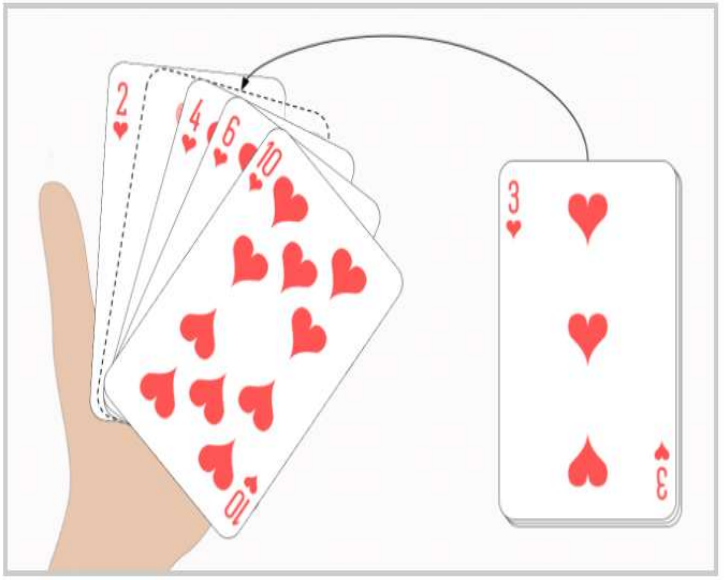
ii. Best Case Time Complexity [Omega] : **$\Omega(n^2)$**

iii. Average Time Complexity [Theta] : **$\Theta(n^2)$**

iv. Space Complexity : **O(1)**

Insertion Sort

Let us start with a playing card example.



- Imagine **being handed one card at a time.** You take the **first card in your hand.**
- After getting the **second card**, you **sort the second card** to the **left or right of it**.
- After getting the **third card**, the third card is **placed to the left**, in **between** or to **the right**, depending on its size.
- And also, all the following cards are placed in the right position.

Insertion Sort

- An element which is to be 'insert'ed in the list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.
- Consider you have **some cards** in your hand. And they are **arranged in the ascending order of their numbers.**
- If I give you another card, and ask you to **insert** the card, What will you do?, so that the cards in your hand are sorted.
- You will have to **go through each card** from the starting or the back and **find the right position for the new card, comparing it's value with each card.**
- Once you find the right position, **you will insert the card there.**

Insertion Sort Process

Step 1 : The first element in the array is assumed to be sorted. Take the second element and store it separately in key.

Step 2 : Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element otherwise leave it.

Step 3 : Then, we make the third element of the array as key and will compare it with elements to it's left and insert it at the right position.

Step 4 : we go on repeating this, until the array is sorted.

- Initial Array

- Since, 6 < 8

6 will get inserted before 8

- Since, 4 < 6

4 will get inserted before 6

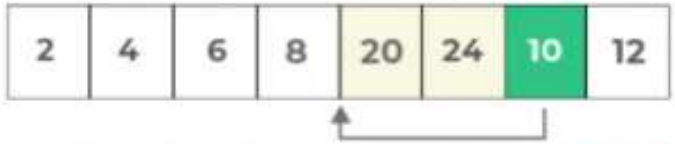
- 20 is at correct position, no insertion needed

- 24 is at correct position, no insertion needed

- Since, 2 < 4

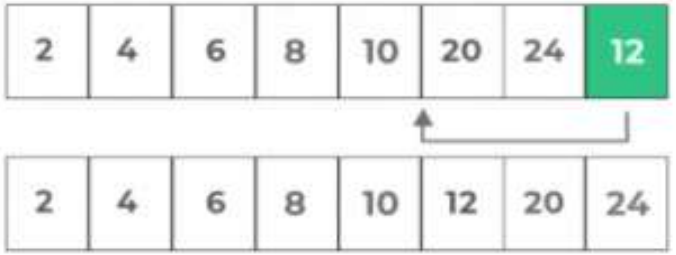
2 will get inserted before 4

- Since, 10 < 20



10 will get inserted before 20

- Since, 12 < 20



12 will get inserted before 20

Insertion Sort Example

Step 1: For i = 1 to n-1

Step 2: key = a[i]

Step 3: j = i - 1

Step 4: while ((key < a[j]) && (j >= 0))

a[j + 1] = a[j]

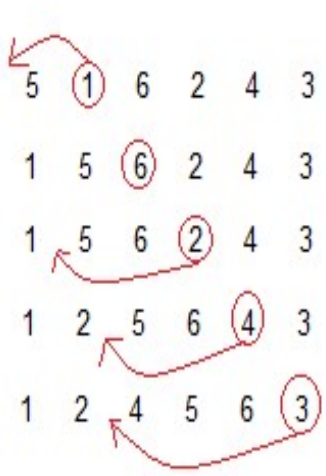
j = j - 1

End While

Step 5: a[j + 1] = key

End For

Lets take this Array.



(Always we start with the second element as key.)

As we can see here, in insertion sort, we pick up a key, and compares it with elemnts ahead of it, and puts the key in the right place

5 has nothing before it.

1 is compared to 5 and is inserted before 5.

6 is greater than 5 and 1.

2 is smaller than 6 and 5, but greater than 1, so its is inserted after 1.

And this goes on...

Insertion Sort

✓See the animation below for more understanding.

Insertion Sort Program

```
#include<stdio.h>
Void insertionsort(int [],int);
main()
{
        int n, i, j,  a[100];
        printf("Enter the size of the list: ");
        scanf("%i", &n);
        printf("Enter  %i  integer values: ", n);
        for (i = 0; i < n; i++)
                scanf("%i", &a[i]);
        Insertionsort(a,n);
        printf("List after Sorting is: ");
        for (i = 0; i < n; i++)
                printf(" %i", a[i]);
}
```

```
Insertionsort(int a[],int n)
{
        int i,key;
        for (i = 1; i < n; i++)
        {
                key = a[i];
                j = i - 1;
                while ((key < a[j]) && (j >= 0))
                {
                        a[j + 1] = a[j];
                        j = j - 1;
                }
                a[j + 1] = key;
        }
}
```

Insertion Sort Worst case Time complexity

Worst Case:

Number of comparisons will be done in the 1st pass (1st Iteration) **: 1**

Number of comparisons will be done in the 2nd pass (2nd Iteration) **: 2**

Number of comparisons will be done in the 3rd pass (3rd Iteration) **: 3**

……

Number of comparisons will be done in the $(n-2)^{th}$ pass **: (n-2)**

Number of comparisons will be done in the $(n-1)^{th}$ pass **: (n-1)**

Total Time = 1+2+3+.....+(n-2)+(n-1)

= n(n-1)/2

=O(n^2)

Hence the **worst case time complexity** of insertion Sort is **O(n^2)**

Insertion Sort Time complexity

Best Case:

If the elements already appear in sorted order, there is only one comparison in the every pass and no swap operation at all.

Number of comparisons will be done in the 1st pass (1st Iteration) **: 1**

Number of comparisons will be done in the 2nd pass (2nd Iteration) **: 1**

Number of comparisons will be done in the 3rd pass (3rd Iteration) **: 1**

……

Number of comparisons will be done in the (n-2)th pass **: 1**

Number of comparisons will be done in the (n-1)th pass **: 1**

Hence ,The **total no of comparisons are:(n-1)**

The **best-case time complexity** of **Insertion Sort is: Ω (n)**

Insertion Sort Time complexity

- ✓Worst Case Time Complexity [Big-O] : $O(n^2)$
- ✓Best Case Time Complexity [Big-omega] : $\Omega(n)$
- ✓Average Time Complexity [Big-theta] : $\Theta(n^2)$
- ✓Space Complexity : $O(1)$

Quick Sort

- The algorithm was developed by a British computer scientist **Tony Hoare in 1959.**
- The name "Quick Sort" comes from the fact that, quick sort is **capable of sorting a list of data elements significantly faster** (twice or thrice faster) than any of the **common sorting algorithms.**
- It is one of the most efficient sorting algorithms and is based on the **splitting of an array** (partition) into **smaller ones and swapping** (exchange) based on the comparison with 'pivot' element
- Quick Sort is also based on the concept of **Divide and Conquer.**
- **Pivot** element can be any element from the array, it can be the **first element**, the **last element** or any **random element.**
- Hence **after** the **first pass**, **pivot** will be **set** at its **position**, with all the elements **smaller** to it on its **left** and all the elements **larger** than to its **right**.

Quick Sort Algorithm

Step 1 : Make any element as pivot

Step 2 : Partition the array on the basis of pivot

Step 3 :Apply quick sort on left partition recursively

Step 4 : Apply quick sort on right partition recursively

Step 1 : Consider the first element of the list as pivot

Step 2 : Define two variables low and high. Set low and high to first and last elements of the list respectively.

Step 3 : Increment low till a[low] <=a[pivot] then stop.

Step 4 : Decrement high till a[high] > a[pivot] then stop.

Step 5 : If (low < high) then swap a[low] and a[high].

Step 6 : Repeat steps 3,4 & 5 until (low > high).

Step 7 : swap the pivot and a[high] element.


List
5 3 8 1 4 6 2 7

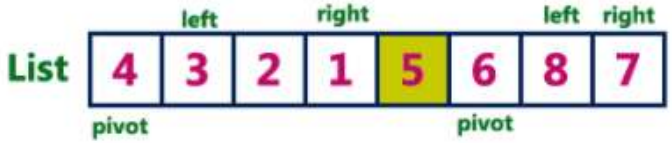

left
right
List
5 3 8 1 4 6 2 7
pivot


left
right
List
5 3 8 1 4 6 2 7
pivot


left
right
List
5 3 2 1 4 6 8 7
pivot

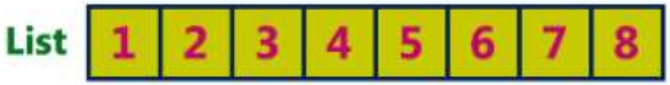

right
left
List
5 3 2 1 4 6 8 7
pivot


List
4 3 2 1 5 6 8 7


left
right
left
right
List
4 3 2 1 5 6 8 7
pivot
pivot

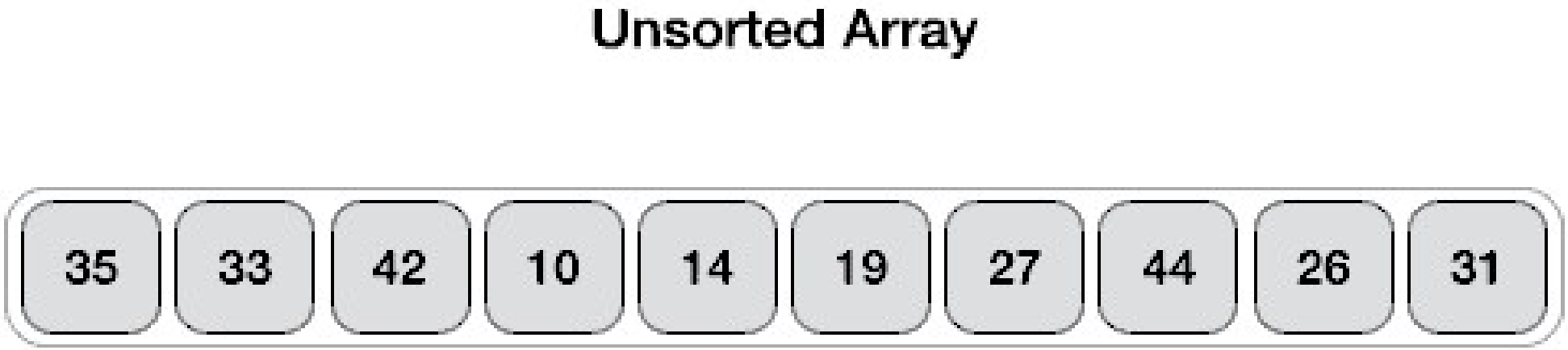

left
right
List
1 3 2 4 5 6 8 7
pivot


left
right
List
1 3 2 4 5 6 8 7
pivot


List
1 2 3 4 5 6 7 8

Quick Sort

✓See the animation below for more understanding.



Quick Sort Program

```
#include <stdio.h>
void quicksort (int [], int, int);
 int main()
{
        int list[50],n, i;
        printf("Enter the number of elements: ");
        scanf("%i", &n);
        printf("Enter the elements to be sorted:\n");
        for (i = 0; i < n; i++)
        scanf("%i", &list[i]);
        quicksort(list, 0, n - 1);
        printf("After applying quick sort\n");
        for (i = 0; i < n; i++)
        printf("%i ", list[i]);
}
```

Quick Sort Program

```
void quicksort(int list[], int low, int high)
{
        int pivot, i, j, temp;
        if (low < high)
        {
        pivot = low;
        i = low;
        j = high;
        while (i < j)
        {
        while (list[i] <= list[pivot] && i <= high)
        {
                i++;
        }
```

```
while (list[j] > list[pivot] && j >= low)
{
        j--;
}
if (i < j)
{
        temp = list[i];
        list[i] = list[j];
        list[j] = temp;
}
}
        temp = list[j];
        list[j] = list[pivot];
        list[pivot] = temp;
        quicksort(list, low, j - 1);
        quicksort(list, j + 1, high);
        }
}
```

Quick Sort Best case Time complexity

Best Case:

$T(n) = T(n/2)+T(n/2) + n$

$T(n) = 2T(n/2) + n$.....1

Substitue (n/2) in place of n in equation 1

$T(n/2) = 2*T(n/4) + (n/2)$.....2

Substitue equation (2) in equation (1)

$= 2*\{ 2*T(n/4) + n/2 \} + n$

$= 2^2*T(n/4) + n + n$

$= 2^2*T(n/4) + 2n$ 3

Substitue (n/4) in place of n in equation 1

$T(n/4) = 2*T(n/8) + (n/4)$.....4

Substitue equation 4 in equation 3

$= 2^2*\{ 2*T(n/8) + (n/4) \} + 2n$

$= 2^3*T(n/8) + 3n$.....5

Observe equ(1) , equ(3) , equ(5) ,

now the next term is :

$= 2^4*T(n/16) + 4n$ and so on....

In Kth pass :

$= 2^k*T(n/(2^k)) + k*n$.....6

Keep going until:

$n/(2^k) = 1$.....7

$n = 2^k$ 8

Apply Log_2 on both side of equation

$\log_2 n = \log_2 2^k$

$\log_2 n = k \log_2 2$ $(\log_2 2 = 1)$

$k = \log_2 n$.....9

Substitue equ(7) and equ(9) in equ(6)

$= 2^k*T(1) + \log^2 n*n$ //T(1)=1 and $2^k = n$ from eq(8)

$= n + \log^2 n*n$

$= n+n*\log^2 n$

So,Best case Time complexity is: $\Omega(n\log^2 n)$

Quick Sort Worst Case Time complexity

- Let's assume that we choose a pivot element in such a way that it gives the **most unbalanced partitions** possible.
- If the **pivot element** is always the **smallest** or **largest** element of the **array** (e.g. because our input data is already sorted)
- The **array** would **not be divided into two** approximately equally sized partitions, one of length 0 and one of length n-1
- Assume total time is T(N)

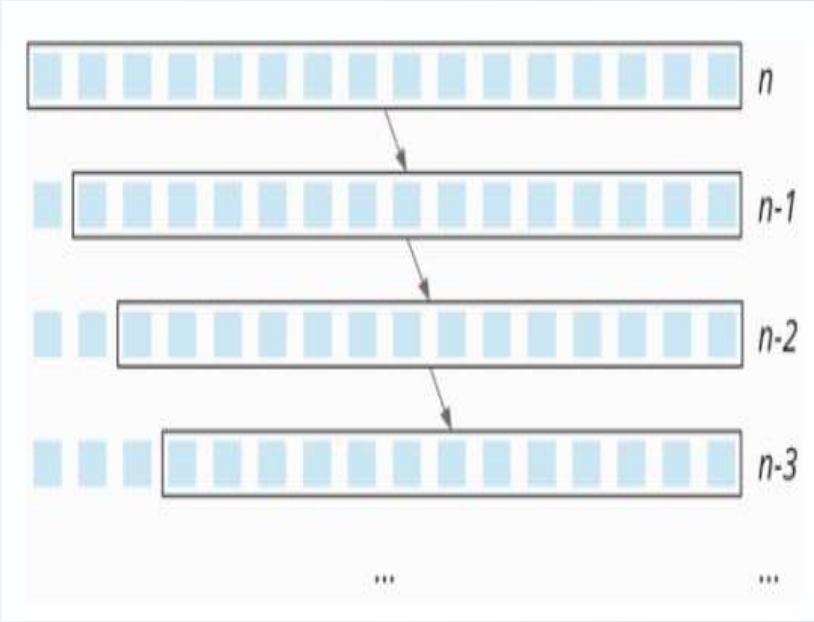
$$T(N) = n + (n-1) + (n-2) + \ldots\ldots\ldots\ldots\ldots + 2 + 1$$

$$= 1 + 2 + \ldots\ldots\ldots\ldots\ldots + (n-2) + (n-1) + n$$

$$= n*(n-1)/2$$

$$= O(n^2)$$

So, Worst case Time complexity is: **$O(n^2)$**



Worst case Time complexity : **$O(n^2)$**

Best case Time complexity : **$\Omega(n \log_2 n)$**

Avg case Time complexity : **$\Theta(n \log_2 n)$**

Merge Sort

- Merge sort is a **divide-and-conquer** algorithm.

- It is based on the idea **of breaking down a list into several sub-lists** until each sub list consists of a single element and merging those sub lists in a manner that results into a sorted list.

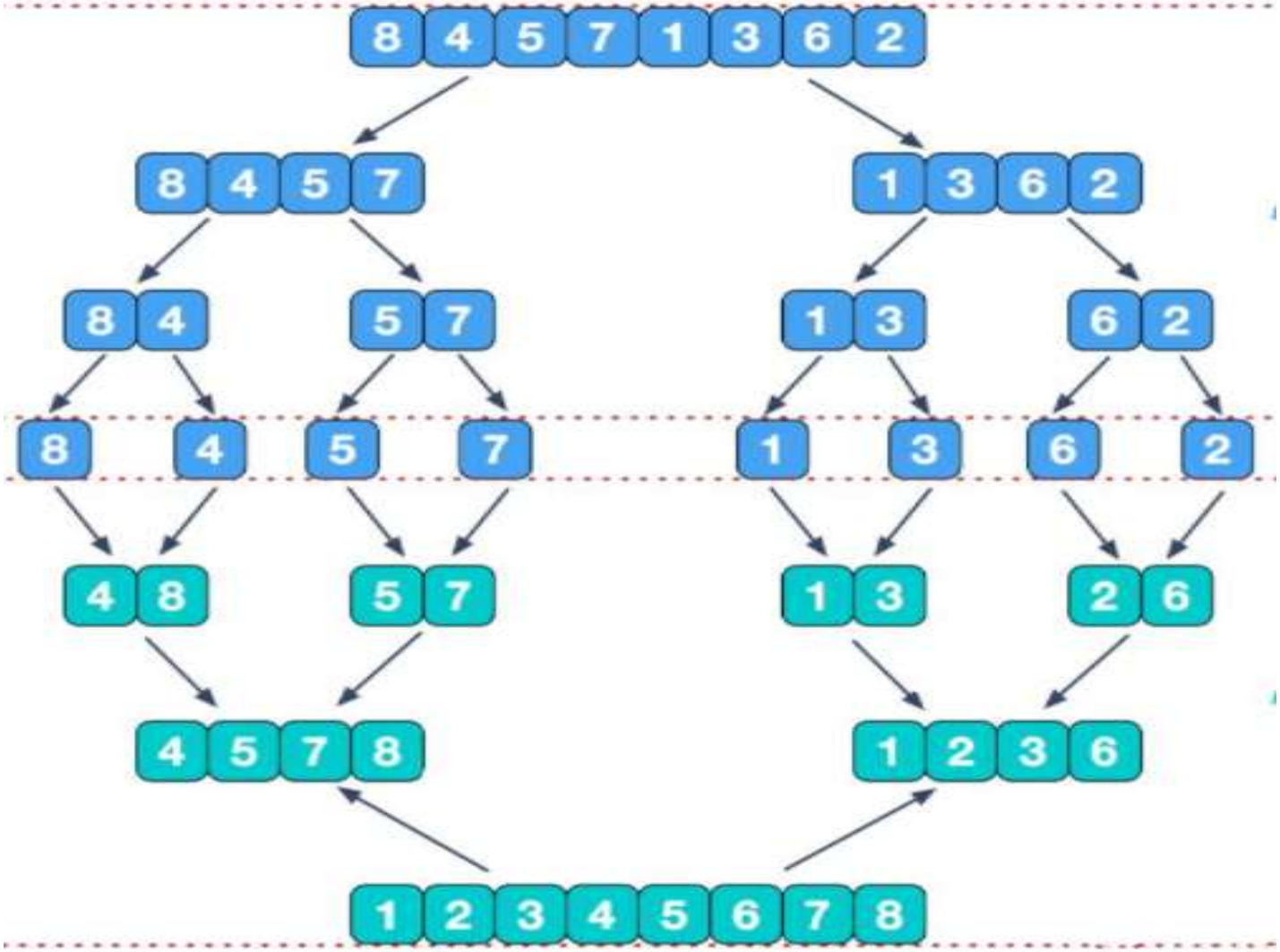
- **Steps :**

    i. Divide **the unsorted list into N sub lists**, each containing 1 element.

    ii. Take adjacent pairs of two lists and merge them to form a list of 2 elements. **N will now convert into N/2 lists of size 2.**

    iii. Repeat the process till **a single sorted list of obtained.**

Merge Sort Example

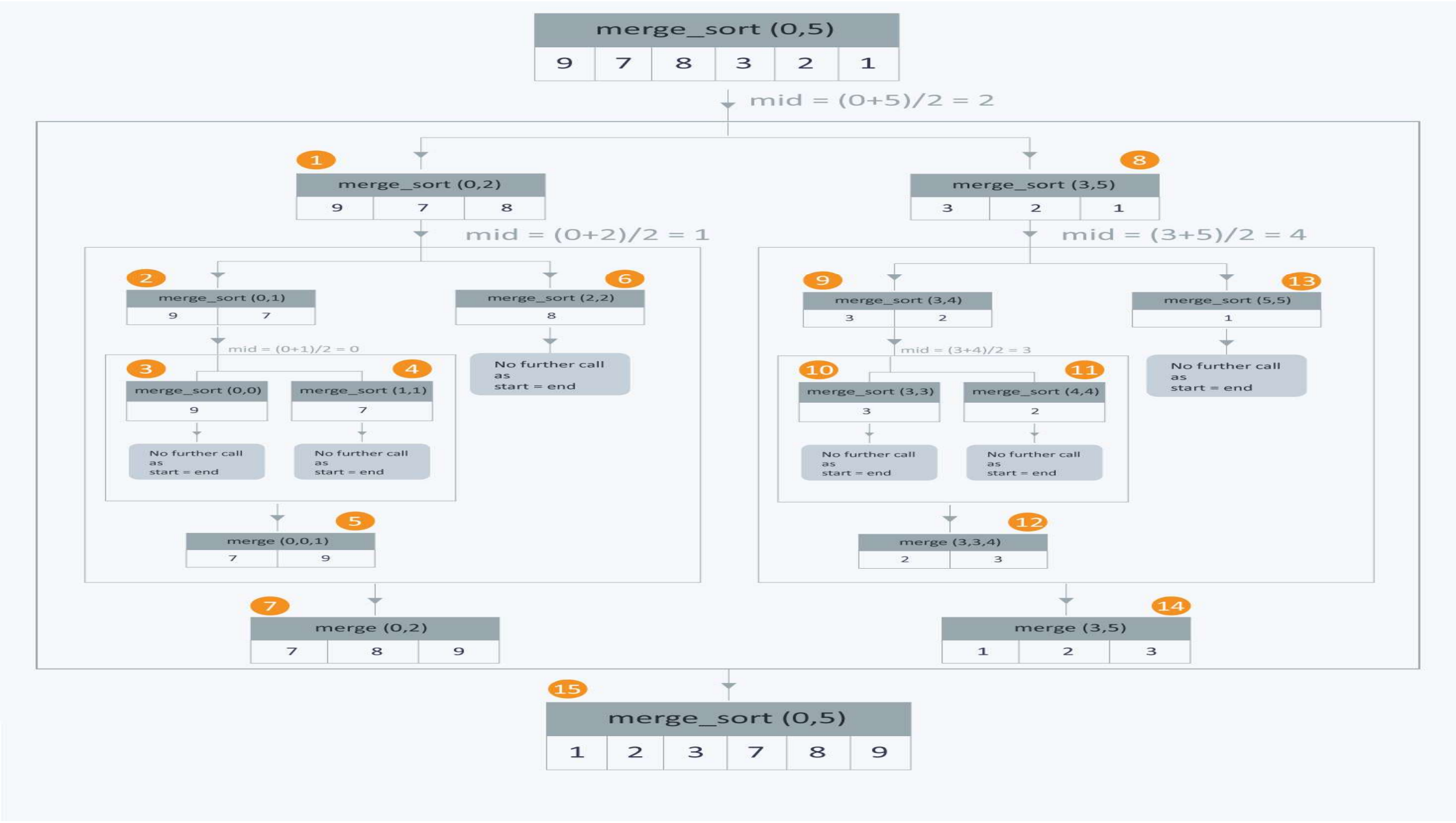


Merge Sort Algorithm

```
MergeSort(arr, low, high):
        if (low > high)
                return
        mid = (low+high)/2
        mergeSort(arr, low, mid)
        mergeSort(arr, mid+1, high)
        merge(arr, low, mid,high)
end
```

Merge Sort

✓See the animation below for more understanding.


merge_sort (0,5)
9 7 8 3 2 1
mid = (0+5)/2 = 2
1
merge_sort (0,2)
9 7 8
mid = (0+2)/2 = 1
8
merge_sort (3,5)
3 2 1
mid = (3+5)/2 = 4
2
merge_sort (0,1)
9 7
mid = (0+1)/2 = 0
6
merge_sort (2,2)
8
No further call as start = end
9
merge_sort (3,4)
3 2
mid = (3+4)/2 = 3
13
merge_sort (5,5)
1
No further call as start = end
3
merge_sort (0,0)
9
4
merge_sort (1,1)
7
No further call as start = end
No further call as start = end
10
merge_sort (3,3)
3
11
merge_sort (4,4)
2
No further call as start = end
No further call as start = end
5
merge (0,0,1)
7 9
12
merge (3,3,4)
2 3
7
merge (0,2)
7 8 9
14
merge (3,5)
1 2 3
15
merge_sort (0,5)
1 2 3 7 8 9

Merge Sort Program

```
#include <stdio.h>
void merge(int [], int, int, int);
void mergesort (int [],int, int);
void main()
{
        int a[50], i, n;
        printf("Enter the list size:");
        scanf("%i", &n);
        printf("\n Enter %i elements:",n);
        for(i = 0; i < n; i++)
                scanf("%i", &a[i]);
        mergesort (a, 0, n - 1);
        printf("\n After merge sort list is:");
        for(i = 0;i < n; i++)
                printf("%i ",a[i]);
}
```

```
void mergesort (int  a[],int low,int high)
{
        int mid;

        if(low < high)
        {
        mid = (low + high) / 2;
        mergesort (a, low, mid);
        mergesort (a, mid + 1, high);
        merge(a, low, mid, high);
        }
}
```

Merge Sort Program

```
void merge(int a[],int low, int mid, int high)
{
        int i,j,k,b[20];
        i = low;
        j = mid + 1;
        k = low;
        while(i <= mid && j <= high)
        {
                if(a[i] <= a[j])
                        b[k++] = a[i++];
                else
                        b[k++] = a[j++];
        }
        while(i <= mid)
                b[k++] = a[i++];
        while(j <= high)
                b[k++] = a[j++];
        for(i = low; i <= high; i++)
                a[i] = b[i];
}
```

Merge Sort Time complexity

Worst Case Time Complexity [Big-O]	: O(n*log n)
Best Case Time Complexity [Omega]	: Ω(n*log n)
Average Time Complexity [Theta]	: Θ (n*log n)
Space Complexity	: O(n)

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Selection Sort	Ω(N^2)	Θ(N^2)	O(N^2)	O(1)
Insertion Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Quick Sort	Ω(N log N)	Θ(N log N)	O(N^2)	O(N)
Merge Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(N)


T
H
A
N
'Q'