ALGORITHMS & DATA STRUCTURES

Dr.Akshita Chanchlani

Name of the Module : Algorithms & Data Structures Using Java.

Prerequisites: Knowledge of programming in C/C++/Java with Object Oriented Concepts.

Weightage : 100 Marks (Theory Exam : 40% + Lab Exam : 40% + Mini Project : 20%).

Importance of the Module:

1. CDAC -Syllabus
2. To improve programming skills
3. Campus Placements
4. Applications in Industry work

Data Structures - Introduction

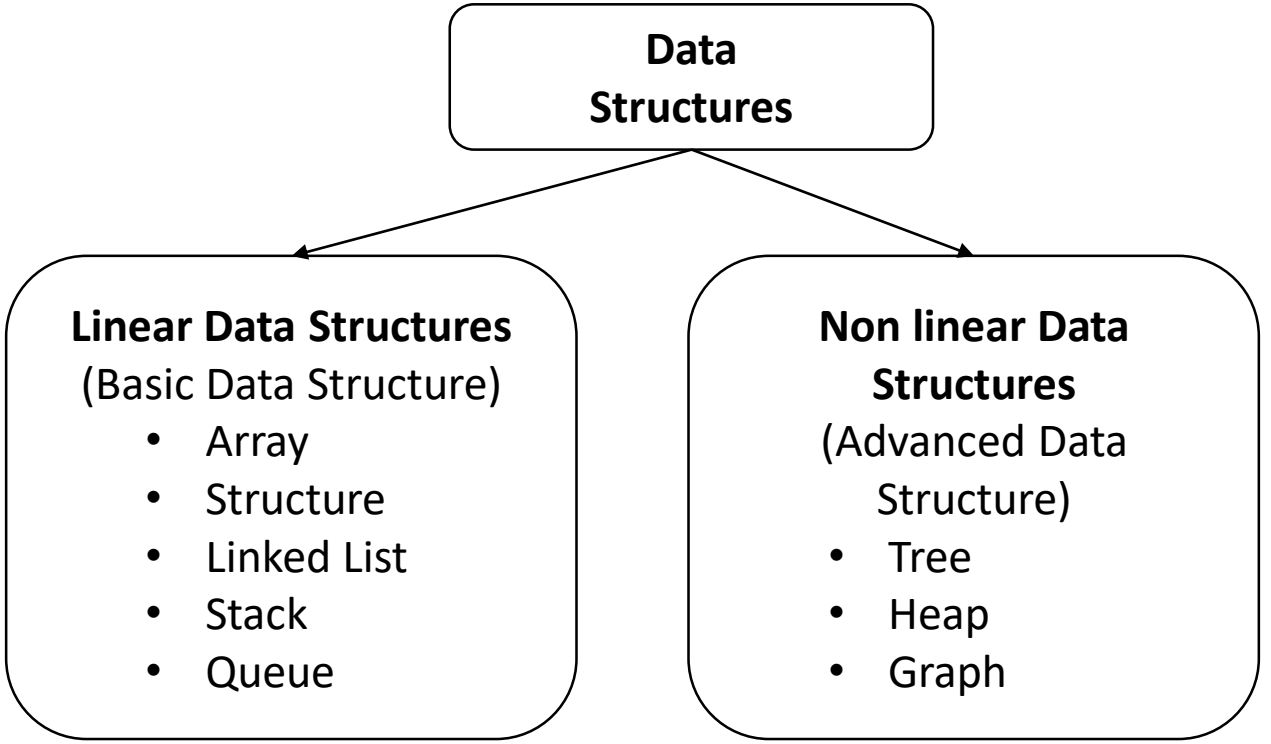
What is Data structure?

- Organising data into memory
- Processing the data efficiently

Why we need Data Structure?

To achieve

1. Efficiency
2. Reusability



Programming Language

- DS and Algorithms are language independent
- We will use **java programming** to implement Data structures

Linear Data Structures

- Data elements are arranged linearly (sequentially) into the memory.
- Data elements can be accessed linearly / Sequentially.

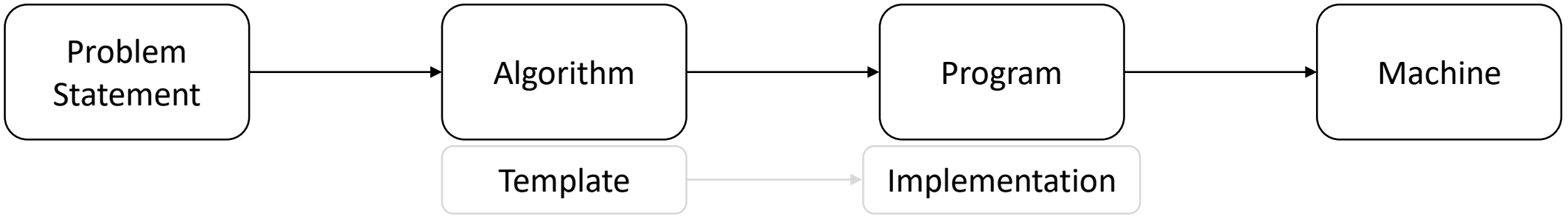
Non linear Data Structures

- Data elements are arranged in non linear manner (hierarchical) into the memory.
- Data elements can be accessed non linearly.

Data Structures - Introduction

- **Algorithm**
  - is a clearly specified set of simple instructions.
  - is a solution to solve a problem.
  - is written in human understandable language.
  - Algorithm is also referred as “pseudo code”.

One problem statement has multiple solutions, out of which we need to select efficient one. Hence we need to do analysis of an algorithm.



e.g. Write an algorithm to find sum of all array elements.
Algorithm:
Step 1: Initialize sum =0
Step 2: Traverse array form index 0 to N-1
Step 3: Add each element in the sum variable
Step 4: Return the final sum

```
Algorithm SumArray(array, size) {
        sum = 0;
        For(index = 0 ; index < size ; index++)
                sum += array[index];
        return sum;
}
```

Types of Data Structure

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

1. Linear /Basic data structures : data elements gets stored /arranged 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 /arranged into the memory in a **non-linear manner** (e.g. hierarchical manner) and hence can be accessed non-linearly.

- **Tree (Hierarchical manner)**
- **Binary Heap**
- **Graph**
- **Hash Table(Associative manner)**

Array, Structure, Union

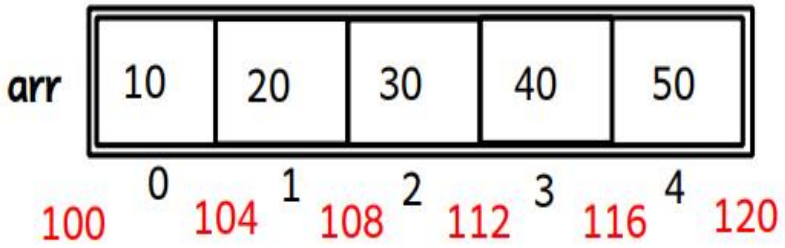
+ **Array:** It is a **basic /linear data structure** which is **a collection /list of logically related similar type of data elements** gets stored/arranged into the memory at **contiguos locations**.

+ **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/arranged into the memory **collectively i.e. as a single entity/record.**

sizeof of the structure = sum of size of all its members.

+ **Union:** Union is same like structure, except, memory allocation i.e. size of union is the size of max size member defined in it and that memory gets shared among all its members for effective memory utilization (can be used in a special case only).

Array



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[0] = *(arr+0) = *(100+0) = *(100) = 10
arr[1] : int --> arr[1] = 20, & arr[1] = 104	arr[1] = *(arr+1) = *(100+1) = *(104) = 20
arr[2] : int --> arr[2] = 30, & arr[2] = 108	arr[2] = *(arr+2) = *(100+2) = *(108) = 30
arr[3] : int --> arr[3] = 40, & arr[3] = 112	arr[3] = *(arr+3) = *(100+3) = *(112) = 40
arr[4] : int --> arr[4] = 50, & arr[4] = 116	arr[4] = *(arr+4) = *(100+4) = *(116) = 50

Structure

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

emp

111	"Akshita Chanchlani"	9999.99
empid	name	salary

100 104 136 140

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

Program and Algorithm

Q. What is a Program?

-A Program is **a finite set of instructions written in any programming language** (either in a high level programming language like C, C++, Java, Python or in a low level programming language like assembly, machine etc...) given to the machine to do specific task.

Q. What is an Algorithm?

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

-**Pseudocode :** It is a **special form of an algorithm**, which is a finite set of instructions written in any human understandable language (like english) **with some programming constraints**, if followed, acomplishesh a given task.

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

Algorithm

 # Algorithm : to do sum of all array elements

Step-1: initially take value of sum is 0.

Step-2: scan an array seqentially from first element max till last element, and add each array element into the sum.

Step-3: return final sum.

 # Pseudocode : 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;
}
```

Types of Algorithms (1. Iterative / Non Recursive)

-There are two types of Algorithms OR there are two approaches to write an algorithm:

1. Iterative (non-recursive) Approach :

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

e.g. iteration

```
for( exp1 ; exp2 ; exp3 ){
   statement/s
}
```

exp1 => initialization
exp2 => termination condition
exp3 => modification

Types of Algorithms (2.Recursive)

2. Recursive Approach:

While writing recursive algorithm -> We need to take care about 3 things

1. **Initialization:** at the time first time calling to recursive function
2. **Base condition/Termination condition :** at the begining of recursive function
3. **Modification:** while recursive function call

Example:

```
Algorithm RecArraySum( A, n, index )
{
   if( index == n )//base condition
         return 0;

   return ( A[ index ] + RecArraySum(A, n, index+1) );
}
```

Recursion

Recursion : it is a process in which we can give call to the function within itself.

function for which recursion is used => recursive function

-there are two types of recursive functions:

1. tail recursive function : recursive function in which recursive function call is the last executable statement.

```
void fun( int n ){
    if( n == 0 )
        return;

    printf("%4d", n);
    fun(n--);//rec function call
}
```

Non-Tail Recursive Function

2. non-tail recursive function : recursive function in which recursive function call is not the last executable statement

```
void fun( int n ){
   if( n == 0 )
      return;

   fun(n--);//rec function call
   printf("%4d", n);
}
```

Algorithm Analysis

- Analysis is done to determine how much resources it require.
- Resources such as time or space
- There are two measures of doing analysis of any algorithm
  - Space Complexity
    - Unit space to store the data into the memory (Input space) and additional space to process the data (Auxiliary space)

    e.g. Algorithm to find sum of all array elements.

    int arr[n] – n units of input space

    sum, index, size – 3 units of auxiliary space

    Total space required = input space + auxiliary space = n + 3 = n units

  - Time Complexity
    - Unit time required to complete any algorithm
    - Approximate measure of time required to complete algorithm
    - Depends on loops in the algorithm
    - Also depends on some external factors like type of machine, no of processed running on machine.
    - That's why we can not find exact time complexity.
  - Method used to calculate complexities, is "**Asymptotic Analysis**"

Asymptotic Analysis

- **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 can be done on the basis of no. of comparisons, in addition of matrices algorithm **addition** is the basic operation and hence on the basis of addition operation analysis can be done.
- It is a study of change in performance of the algorithm, with the change in the order of inputs.
- It is not exact analysis

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

"**Worst case time complexity" :** if an algo takes **maximum** 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 minimum nor maximum** amount of time to run to completion then it is referred as an average case time complexity.

Asymptotic Notations

- Few mathematical notations are used to denote complexities.
- Asymptotic analysis is not exact analysis

Big' O notation / Big – Oh notation (O)
Represents upper bound of the running algorithm
It is used to indicate the worst case complexity of an algorithm

- **Space complexity**

Unit space to store the data (Input space) and additional space to process the data (Auxiliary space). O(1), O(n), O(n^2)

- **Time complexity**

Unit time required to complete any algorithm. Approximate measure of time required to complete any algorithm.
Depends on loops in the algorithm. O(n^3), O(n^2), O(n log n), O(n), O(log n), O(1)

Omega notation (Ω)
Represents lower bound of the running algorithm. It is used to indicate the best case complexity of an algorithm

Theta notation (Θ)
Represents upper and lower bound of the running time of an algorithm (tight bound)
It is used to indicate the average case complexity of an algorithm

Space Complexity

- Space Complexity of an algorithm is the amount of space i.e. computer memory it needs to run to completion.
- It is the unit space required to execute the algorithm.
- Space Complexity = Code Space + Data Space + Stack Space (applicable only for recursive algo)
- Code Space = space required for an instructions
- Data Space = space required for simple variables, constants & instance variables.
- Stack Space = space required for function activation records (local vars, formal parameters, return address, old frame pointer etc...).

- - Space Complexity has two components:
- 1. Fixed component : code space and data space (space required for simple vars & constants).
- 2. Variable component : data space for instance characteristics (i.e. space required for instance vars) and stack space (which is applicable only in recursive
- algorithms).

Space Complexity

Total Space = Input Space + Auxiliary Space

Input Space = Space required to store input (Actual Data Structure)
Auxiliary Space = Space required to process the given input

Space Complexity of Algorithm

Calculation of Space complexity of non-recursive algorithm:

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

Total Space = Input Space + Auxiliary Space

A[n] = n units of input space
Sum , index = 2 units of auxiliary space

Total Space = n+2 units

Space Complexity Example1

```
type addition(type num1,type num2)
{
     type res = num1+num2;
    return res;
}
```

Input Space = 2 units (num1, num2)
Auxiliary Space = 1 unit (res)

Total Space = 3 units
Total Space α 1 unit
Space Complexity = O(1)

NOTE : Irrespective of data type and value of variable , space required is constant.
To indicate complexity we use O() notation.

Space Complexity Example2

```
type print_Arr(int arr[],int size)
{
        int i;
        // traverse array from first to size-1 elements
        //  print array
}
```

Space Complexity Example2 Solution

```
type print_Arr(int arr[],int size)
{
        int i;
        // traverse array from first to size-1 elements
        //  print array
}
```

Input Space = n units (assume array is of “n” elements)
Auxiliary Space = 2 unit (i,size)

Total Space = n+2 units //assume n >>>>>>
Total Space α n unit
Space Complexity = O(n)

Space Complexity Example3

```
type print_Arr(int arr[],int row,int col)
{
        int I,j;
      //outer for loop
       for(i=0;i<row;i++)
        {
            for(j=0;j<col;j++)
            {
                        ……
            }
      }
         //  print array
}
```

Space Complexity Example3 Solution

```
type print_Arr(int arr[],int row,int col)
{
        int I,j;
      //outer for loop
       for(i=0;i<row;i++)
        {
            for(j=0;j<col;j++)
            {
                        ……
            }
      }
         //  print array
}
```

Input Space = n*n units (assume array is of “n*n” size)

Input Space = n^2 unit

Auxiliary Space = 4 elements (I,j,row,col)

Total Space = n^2 + 4 units //assume n >>>>>>

Total Space α n^2 unit

Space Complexity = $O(n^2)$

FAR (Function Activation Record)

- - When any function gets called, one entry gets created onto the stack for that function call, referred as function activation record / stack frame, it contains formal params, local vars, return addr, old frame pointer etc...

Time Complexity

```
Statement;
```

constant

```
for(i=0; i< n; i++)
{
        statements;
}
```

Linear

```
for(i=0; i< n; i++)
{
        for(j=0; j< n; j++)
        {
                statements;
        }

}
```

Quadratic

```
for(i=n; i>0; i/=2)
{
        statement
}
```

Logarithmic

Time Complexity

 # Time Complexity:

Time Complexity = Compilation Time + Execution Time

Time complexity has two components :

1. Fixed component : compilation time

2. Variable component : execution time => it depends on instance characteristics of an algorithm.

Example :

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

   return sum;
}
```

Time complexity Examples

- **Unit time required to complete the algorithm.**
- **It gives always approximate measure.**
- **It is based on how many iterations that algorithm is taking**

1. Write a program to calculate factorial of given number.

2. Print 2-D matrix of n x n.

3. Print given number into binary.

4. Print table of given number.

Asymptotic Notations

 # Asympotic Notations:

It does not talk about exact analysis, it is presented in terms of number of elements. That's called as Big'O notation.

1. Big Omega (Ω) : this notation is used to denote **best case time complexity** – also called as **asymptotic lower bound,** running time of an algorithm cannot be less than its asymptotic lower bound.

2. Big Oh (O) : this notation is used to denote **worst case time complexity** -also called as **asymptotic upper bound,** running time of an algorithm cannot be more than its asymptotic upper bound.

3. Big Theta (θ) : this notation is used to denote an **average case time complexity** - also called as **asymptotic tight bound,** running time of an algorithm cannot be less than its asymptotic lower bound and cannot be more than its asymptotic upper bound i.e. it is **tightly bounded.**

Asymptotic Analysis

-for size of an array = 5 => instruction/s inside for loop will execute 5 no. of times
-for size of an array = 10 => instruction/s inside for loop will execute 10 no. of times
-for size of an array = 20 => instruction/s inside for loop will execute 20 no. of times
-for size of an array = **n** => instruction/s inside for loop will execute **"n"** no. of times

Scenario-1 :

Machine-1 : Pentium-4 : Algorithm : input size = 10
Machine-2 : Core i5 : Algorithm : input size = 10

Scenario-2 :

Machine-1 : Core i5 : Algorithm : input size = 10 : system fully loaded with other processes
Machine-2 : Core i5 : Algorithm : input size = 10 : system not fully loaded with other processes.

-It is observed that, **execution time is not only depends on instance characteristics**, it also depends on **some external factors** like hardware on which algorithm is running as well as other conditions, and hence it is not a good practice to decide efficiency of an algo i.e. calculation of time complexity on the basis of an execution time and compilation time, hence to do analysis of an algorithms **asymptotic analysis** is preferred.

Time Complexity : Approximate time required to complete the algorithm.

$O(1)$ = Constant time

$O(n)$ = time $\propto n$

$O(\log n)$ = time $\propto \log n$

$O(n \log n)$ = time $\propto n \log n$

$O(n^2)$ = time $\propto n^2$

Searching Algorithm : Linear Search

- Search a number in a list of given numbers (random order)

- **Algorithm**
  - Step 1: Accept key from user
  - Step 2: Traverse list from start to end
  - Step 3: Compare key with each element of the list
  - Step 4: If key is found return true else false

```
Algorithm linear_search(a, size, key)
{
        for(i = 0 ; i <= size ; i++ )
        {
                if(a[i] == key)
                        return true;
        }
        return false;
}
```

Searching Algorithms

Best Case: If key element is found at very first position in only 1 comparison then it is considered as a best case and running time of an algorithm in this case is **O(1)** => hence time complexity of linear search algorithm in base case = **Ω(1).**

Worst Case: If either key element is found at last position or key element does not exists, in this case 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)** => hence time complexity of linear search algorithm in worst case = **O(n).**

Average Case: If key element 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) => O(n) => hence time complexity = **θ(n).**

Binary Search /Logarithmic Search /Half Interval 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 value of key element from the user which is to be search.

Step-2: In first iteration, find/calculate **mid position** by the formula **mid=(left+right)/2**, (by means of finding mid position big size array gets divided logically into 2 subarrays, left subarray and right subarray, **left subarray => [left to mid-1]** & **right subarray => [mid+1 to right]**.

Step-3 : Compare value of key element with an element which is at mid position, **if key matches in very first iteration in only one comparison then it is considered as a best case**, if key matches with mid pos element then return true otherwise if key do not matches then we have to go to next iteration, and in next iteration we go to search key either into the left subarray or into the right subarray.

Step-4 : Repeat Step-2 & Step-3 till either key is found or max till subarray is valid, **if subarray is not valid then key is not found in this case return false.**

Searching Algorithm : Binary Search

- Given an integer x and integers A0, A1, …An-1, which are pre-sorted and already in memory, find i such that Ai = x or return i = -1 if x is not in the input

- **Algorithm**
  - Step 1: Accept key from user
  - Step 2: Check if x is the middle element. If so x is found at mid
  - Step 3: If x is smaller than the middle element, apply same strategy to the sorted subarray to the left of middle element.
  - Step 4: If x is larger than the middle element, apply same strategy to the sorted subarray to the right of middle element

Binary Search

-As in each iteration 1 comparison takes place and search space is getting reduced by half.

n => n/2 => n/4 => n/8 .....

after iteration-1 => n/2 + 1 => **$T(n) = (n/2^1) + 1$**

after iteration-2 => n/4 + 2 => **$T(n) = (n/2^2) + 2$**

after iteration-3 => n/8 + 3 => **$T(n) = (n/2^3) + 3$**

Lets assume, after **k** iterations => **$T(n) = (n/2^k) + k$ ..... (equation-I)**

let us assume,

=> **$n = 2^k$**

=> log n = log 2^k (by taking log on both sides)

=> log n = k log 2

=> log n = k (as log 2 ~= 1)

=> **k = log n**

By substituting value of **$n = 2^k$ & k = log n** in **equation-I,** we get

=> $T(n) = (n / 2^k) + k$

=> $T(n) = (2^k / 2^k) + \log n$

=> T(n) = 1 + log n => T(n) = O(1 + log n) => **$T(n) = O(\log n)$.**

Binary Search

```
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;
}
```

Searching Algorithms : Time Complexity

Linear Search :

	No of Comparisons		Running Time	Time Complexity
Best Case	1	Key found at very first position	O(1)	O(1)
Average Case	n/2	Key found at in between position	O(n/2) = O(n)	O(n)
Worst Case	n	Key found at last position or not found	O(n)	O(n)

Binary Search :

	No of Comparisons		Running Time	Time Complexity
Best Case	1	Key found in very first iteration	O(1)	O(1)
Average Case	log n	Key found at non-leaf position	O(log n)	O(log n)
Worst Case	log n	if either key is not found or key is found at leaf position	O(log n)	O(log n)

Thank you!

Dr.Akshita Chanchlani

<akshita.chanchlani@sunbeaminfo.com>