Module Name : Algorithms & Data Structures Using Java.
==
DAY-01:

Introduction to data structures?

Q. Why there is a need of data structures?

- if we want to store marks of 100 students

```
int m1, m2, m3, m4, ....., m100;//400 bytes if
sizeof(int) = 4 bytes

int marks[ 100 ];//400 bytes - if sizeof(int) = 4 bytes
```

- we want to store rollno, marks & name

```
rollno   : int
marks    : float
name     : char [] / String / String

struct student
{
    int rollno;
    char name[ 32 ];
    float marks;
};

struct student s1;

class Employee
{
    //data members
    int empid;
    String empName;
    float salary;
    //member functions/methods
};

Employee e1;
Employee e2;
```

=> to learn data structures is not learn any programming language, it is a programming concept i.e. it is nothing but to learn algorithms, and algorithms learned in data structures can be implemented by using any programming language.

 # algorithm to do sum of array elements => end user (human being)
step-1: initially take sum as 0.
step-2: traverse an array and add each array element into sum sequentially
from first element max till last element.
step-3: return final sum.

 # pseudocode => programmer user

```
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;
}
```

 # program => compiler => machine

```
int arraySum(int [] arr, int size){
     int sum = 0;
     for( int index = 0 ; index < size ; index++ ){
          sum += arr[ index ];
     }
     return sum;
}
```

Bank Manager => Algorithm => Project Manager => Software Architect
=> Pseudocode => Developers => Program => Machine

Problem : **"Searching"** => to search / to find an element (can be referred as a
key element) into a collection/list of elements.

1. Linear Search
2. Binary Search

Problem : **"Sorting"** => to arrange data elements in a collection / list of elements either in an ascending order or in a descending order.

1. Selection Sort
2. Bubble Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort

etc....

- when one problem has many solutions we need to go for an efficient solution.

City-1:
City-2:

multiple paths exists => optimum path
distance, condition, traffic situation ....

- to traverse an array => to visit each array element sequentially from first element max till last element.

- there are two types of algorithms:

1. iterative approach (non-recursive)
2. recursive approach

- recursion
- recursive function
- recursive function call
- tail-recursive function
- non-tail recursive function

```
Class Employee
{
    int empid;
    String name;
    float salary;
```

};

- object e1 is an instance of Employee class
Employee e1(1, "sachin", 1111.11);
Employee e2(2, "nilesh", 2222.22);
Employee e3(3, "amit", 3333.33);

 # Space Complexity:
for size of an array = 5 => index = 0 to 5 => only 1 mem copy of index = 1 unit
for size of an array = 10 => index = 0 to 10 => only 1 mem copy of index = 1 unit
.
.
for size of an array = n => index = 0 to n => only 1 mem copy of index = 1 unit

for any input size array we require only 1 memory copy of index var =>
simple var

+ sum:
for size of an array = 5 => sum => only 1 mem copy of sum = 1 unit
for size of an array = 10 => sum => only 1 mem copy of sum = 1 unit
.
.

for any input size array we require only 1 memory copy of sum var => simple var

+ n = input size of an array -> instance characteristics of an algo

for size of an array = 5 => if n = 5 => 5 memory copies required to store 5 ele's => 5 units

for size of an array = 10 => if n = 10 => 10 memory copies required to store 10 ele's => 10 units for size of an array = 20 => if n = 20 => 20 memory copies required to store 20 ele's => 20 units

for size of an array = 100 => if n = 100 => 100 memory copies required to store 100 ele's => 100 units

size

- for any input size array no. of instructions inside an algo remains fixed i.e. space required for instructions i.e. code space for any size array will going to remains fixed or constant.

```
int sum( int n1, int n2 )//n1 & n2 are formal params
{
    int res;//local var
    res = n1 + n2;
    return res;

}
```

- When any function completes its excution control goes back into its calling function as an addr of next instruction to be executed in its calling function gets stored into FAR of that function as a **"return addr"**.

FAR contains:
local vars
formal params
return addr => addr of next instruction to be executed in its calling function
old frame pointer => an addr of its prev stack frame/FAR.
etc...

Linear Search:

```
for( index = 1 ; index <= n ; index++ ){
    if( key  == arr[ index ] )
        return true;
}

return false;
```

- In Linear Search best case "if key found at very first position"

for size of an array = 10, no. of comparisons = 1
for size of an array = 20, no. of comparisons = 1
for size of an array = 50, no. of comparisons = 1
.
.

for any input size array => no. of comparisons = 1
Running Time => O(1)

- In Linear Search worst case "if either key found at last first position or key does not exists"

for size of an array = 10, no. of comparisons = 10
for size of an array = 20, no. of comparisons = 20
for size of an array = 50, no. of comparisons = 50
.
.

for size of an array = n, no. of comparisons = n
Running Time => O(n)

Lab Work => Implement Linear Search => by non-rec as well rec way

DAY-02:
- **linear search**
- **binary search**
- **comparison between searching algo**
- **sorting algorithms:**

basic sorting algo’s : selection, bubble & insertion

assumption-1:
if running time of an algo is having any additive / substractive / divisive / multiplicative constant then it can be neglected.

e.g.
O(n + 3) => O(n)
O(n - 2) => O(n)
O(n / 5) => O(n)
O(6 * n) => O(n)

Binary Search:

by means of calculating mid position big size array gets divided logically into two subarray’s:
left subarray => left to mid-1
right subarray => mid+1 to right

for left subarray => value of left remains same, whereas value of right = mid-1

for right subarray => value of right remains same, whereas value of left = mid+1

best case occurs in binary search if key is found in very first iteration in only 1 comparison.
if size of an array = 10, no. Of comparisons = 1
if size of an array = 20, no. Of comparisons = 1
.
.
for any input size array, no. Of comparisons = 1
=> running time => O(1)

- in this algo, **in every iteration 1 comparison takes place and search space gets divided by half** i.e. array gets divided logically into two subarray's and in next iteration we will search key either into left subarray or into right subarray.

 # Substitution method to calculate time complexity of binary search:

Size of an array = 1000
1000 = n
500 = n/2 (search space reduced by half)
250 = n/4 (search space reduced by one\fourth)
125 = n/8
.
.
.

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

for iteration-1 input size of an array => n
after iteration-1: n/2 + 1 => $n / 2^1 + 1$ (comparisons)
after iteration-2: n/4 + 2 => $n / 2^2 + 2$ (comparisions)
after iteration-3: n/8 + 3 => $n / 2^3 + 3$
.
.
.
after iteration-k: $n / 2^k + k$ .... eq-I

$T(n) = n / 2^k + k$ .... eq-I

lets assume, $n = 2^k$
$n = 2^k$
$\log n = \log 2^k$ .... **[by taking log on both sides]**
$\log n = k \log 2$

log n = k .... **[as log 2 ~= 1]**
k = log n

put value of $n = 2^k$ and k = log n in eq-I, we get
$T(n) = n / 2^k + k$
=> $T(n) = 2^k / 2^k + \log n$
=> $T(n) = 1 + \log n$
=> $T(n) = O(1 + \log n)$
=> $T(n) = O(\log n + 1)$
=> $\underline{T(n) = O(\log n)}$.

1. Selection Sort:

total no. of comparisons = (n-1)+(n-2)+(n-3) + ...+ 1
=> n(n-1) / 2
hence
=> T(n) = O(n(n-1) / 2)
=> $T(n) = O((n^2 - n) / 2)$
=> $T(n) = O(n^2 - n)$
=> **$T(n) = O(n^2)$**

assumption:
if running time of an algo is having a ploynomial then in its time complexity only leading term will be considered.
e.g.
$O(n^3 + n^2 + 5) \Rightarrow O(n^3)$.

assumption:
if an algo contains a nested loops and no. Of iterations of outer loop and inner loop dont know in advanced then running time of such algo will be whatever time required for statements which are inside inner loop.

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

    for( j = 0 ; j < n ; j++ ){
        statement/s  => n*n no. Of times => O(n^2) times
    }

}
```

+ features of sorting algorithms:
1. inplace => if a sorting algo do not takes extra space (i.e. space required other than actual data ele's and constant space) to sort data elements in a collection/list of elements.

2. adaptive => if a sorting algorithm works efficiently for already sorted input sequence then it is referred as an adaptive.

3. stable => if in a sorting algorithm, relative order of two elements having same key value remains same even after sorting then such sorting algorithm is referred as stable.

Input array => 10 40 20 30 10' 50

After Sorting:
Output => 10 10' 20 30 40 50 => stable

Output => 10' 10 20 30 40 50 => not stable

 # Design & Analysis of an Algorithm By Coreman

2. Bubble Sort: