Data structures and Algorithms

Data structure

- Orgnising data into memory for efficient processing
- And we can perform operations like add, delete, search or sort on that structure
- Abstract Data types
- There are two types of Data structure

Linear (Basic)

1. Array
2. Structutre/class
3. Stack
4. Queue
5. Linked List

Non Linear (Advanced)

1. Tree
2. Heap
3. Graph

Algorithm

- step by step solution to given problem statement
- set of instructions to human (engineer / developer)
- it is written into human understandable language
- is programming language independant
- it is like template/blue print

to achieve

1. Abstraction
2. Reusability
3. Efficiency - (Time/ space)

Algorithm Analysis

Time Analysis / Time Complexity

Exact Analysis

- actual running time of your algorithm
- It is dependent on processor type, size of data structure, no of processes running

Approximate Analysis

- Asymptotic Analysis (mathematical way of finding time and space)
- time required is number of iterations in your algorithm
- time is directly proportional to no of interations/ no of element
- 'Big O' notation is used to indicate complexities

1. Print array on console

```
void printArray(int[] arr, int n){
        for(int i = 0 ; i < n ; i++)
                sysout(arr[i]);
}
```

No. of iterations = n

Time $\propto$ no. of iterations

Time $\propto n$

$T(n) = O(n)$

2. Print 2D array on console

```
void print2DArray(int[] arr, int n, int m){
        for(int i = 0 ; i < n ; i++){
                for(int j = 0 ; j < m ; j++)
                        sysout(arr[i][j]);
        }
}
```

No. of iterations = $n * m$

$= n * n$

$= n^2$

Time $\propto n^2$

$T(n) = O(n^2)$

3. find sum of two numbers

```
int sumOfTwoNumbers(int num1, int num2){
        return num1 + num2;
}
```

Time required constant

Time $\propto 1$

$T(n) = O(1)$

4. Find binary of decimal number

```
void printBinary(int n){
        int i = n;
        while( i > 0 ){
                int rem = i % 2;
                sysout(rem);
                i = i / 2;
        }
}
```

10 > 0 :
10 % 2 -> 0
10 / 2
5 > 0 :
5 % 2 -> 1
5 / 2
2 > 0 :
2 % 2 -> 0
2 / 2
1 > 0 :
1 % 2 -> 1
1 / 2
0 > 0 : false

$i = n, n/2, n/4, \frac{n}{8} \ldots\ldots$

$i = \frac{n}{2^0}, \frac{n}{2^1}, \frac{n}{2^2}, \frac{n}{2^3} \ldots \frac{n}{2^{itr}}$

For $i = 1$ last time condⁿ will be true

$$\frac{n}{2^{itr}} = 1$$

$$n = 2^{itr}$$

$$\log n = \log 2^{itr}$$

$$itr * \log 2 = \log n$$

$$itr = \frac{\log n}{\log 2}$$

$$\text{Time} \propto \frac{\log n}{\log 2}$$

$$T(n) = O(\log n)$$

Time Compexities - O(1), O(log n), O(n), O(n logn), O(n^2), O(n^3), ..., O(2^n)

T(n) - Mathematics polynomial

modification - '+'/'-' -> T(n) will be in terms of n
modification - '*'/'/' -> T(n) will be in terms of log n

```
for(i=n; i>0; i=i/2) -> i=10,5,2,1,0 -> log(n)
for(i=1, i<n; i=i*2) -> i=1,2,4,8,16 -> log(n)

for(i=0; i<n; i++) -> O(n)
for(i=n; i>0; i--) -> O(n)
for(i=0; i<n, i=i+20) -> O(n)
for(i=1; i<=10; i++) -> O(1)
```

Space Complexity

- it is space required to execute in memory
- total space is addition of input space and auxillary space

total space = input space + auxillary space

input space - space required to store actual data
auxillarhy space - space required to process actual data

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

Input space = n

Auxillary space = 3
(size, i, sum)

Total space = n + 3

space $\propto$ n

$S(n) = O(n)$

S(n) - O(1), O(n), O(n^2), O(n^3)......

Searching

- finding data from collection of data
- There are two searching algorithms
    1. Linear search
    2. Binary search

Linear Search

Key = 88	Best case	O(1)
Key = 11	Average case	O(n)
Key = 14	Worst case	O(n)
Key = 100		

Binary Search

Key = 55	Best case	O(1)
Key = 77	Average case	O(log n)
Key = 99	Worst case	O(log n)
Key = 100		

Approach

Iterative

implemented using loops

```
int factorial(int num){
        int fact = 1;
        while(num){
                fact *= num;
                num = num - 1;
        }
        return fact;
}
```

Time complexity = no of iterations

$T(n) = O(n)$

Recursive

Recursive functions

$n! = n * (n-1)!$

if $n = 0$, stop

```
int recFactorial(int num){
        if(num == 0)
                return 1;
        return num * recFactorial(num-1);
}
```

Time complexity = no of recursive function calls

$T(n) = O(n)$

```
int main(void)
{
        int fact = rFact(5);
        return 0;
}
```

```
int rFact(int num){
        if(num == 0)
                return 1;
        return num * rFact(num-1);
}
```

```
int rFact(5){
        if(5 == 0)
                return 1;
        return 5 * rFact(4);
}
```

5 * 24

120

```
int rFact(4){
        if(4 == 0)
                return 1;
        return 4 * rFact(3);
}
```

4 * 6

24

```
int rFact(3){
        if(3 == 0)
                return 1;
        return 3 * rFact(2);
}
```

3 * 2

6

```
int rFact(0){
        if(0 == 0)
                return 1;
        return 5 * rFact(4);
}
```

```
int rFact(1){
        if(1 == 0)
                return 1;
        return 1 * rFact(0);
}
```

1 * 1

1

```
int rFact(2){
        if(2 == 0)
                return 1;
        return 2 * rFact(1);
}
```

2 * 1

2

Sorting

- arrangement of elements in collection either asceding or descending order of their values

Basis:

1. **Selection sort**
2. **Bubble sort**
3. **Insertion sort**

Advanced:

4. **Merge sort**
5. **Quick sort**
6. **Heap sort**