Data Structures and Algorithms

Data Structure

- Data structure is organising data into memory for efficient access.
- on which we can perform various operations like add, remove, search, sort and update etc
- Abstract Data Types
- There are two types of Data structures

1. Linear Data structure (basic)
   - data is arranged linearly (sequentially)
   1. Array
   2. Structure / class
   3. Queue
   4. Stack
   5. Linked List

2. Non Linear Data structure (Advanced)
   - data is arranged non-linearly (heirachical)
   1. Tree
   2. Graph

Algorithm

- step by step solution to the given problem statement
- set of instructions to the human (programmer/developer)
- template ---> program

eg. Find sum of array elements

step 1 - create variable to store sum and initialize it with zero
step 2 - traverse the array from 0th index to size-1 index
step 3 - add every element of array into sum variable
step 4 - return /print the final sum

eg. searching, sorting

to achieve

- Abstraction
- Reusability
- Efficiency (time and space)

Traversing - visiting each and every element of collection/structure at least once
Analysis - time and space measurment of data structure and algorithm

Algorithm Analysis

Exact Analysis

- exact time and spce required to execute
- space is dependent on no of elements and type of element
- time is dependent on type of machine, no of proecesses running at that time, size of data structure

Approximate Analysis

- approximate budget/measurement of time and space
- Asymptotic Analysis - mathematical approach to find time and space requirement of algorithm
- observing change in behavior of algorithm on change in input
- "Big O" / "O()" notation is used to indicate time and space

Time Measurement / Time Complexity

- time complexity is approximate time analysis
- to execute single statement will need 1 unit of time
- no of iterations of the loops are considered
- time is directly proportional to the no of iterations of the loop

1. Print 1D array

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

No. of iterations of loop = n

Time Required ∝ No. of Iterations

Time ∝ n unit

Time = n

T(n) = O(n)

2. Print 2D array

```
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]);
}
```

iterations (outer loop) $r1 = n$

iterations (inner loop) $r2 = m = n$

Total iterations $= n * n = n^2$

Time $\propto n^2$

Time $= n^2$

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

3. Add two numbers

```
int addTwoNumbers(int num1, int num2){
        int res = num1 + num2;
        return res;
}
```

irrespective of input values, this algorithm is taking some constant amount of time, it will be denoted as

$T(n) = O(1)$

4. print table of given number

```
void printTable(int num){
        for(int i = 1 ; i <= 10 ; i++)
                sysout(i * num);
}
```

irrespective of input value, loop is going to iterate 10 time, so constant time will be needed.

$T(n) = O(1)$

5. Print binary of given decimal

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

```
num = 9
        9 > 0:
                9 % 2  -> 1
                9 / 2 -> 4
        4 > 0:
                4 % 2 -> 0
                4 / 2 -> 2
        2 > 0:
                2 % 2 -> 0
                2 / 2 -> 1
        1 > 0:
                1 % 2 -> 1
                1 / 2 -> 0
```

$i = 9, 4, 2, 1, 0$ (✓ ✗)

$i = n, n/2, n/4, n/8 \dots$

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

for $i=1$, last time loop condn is true.

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

$$n = 2^{itr}$$

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

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

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

$$Time \propto \frac{1}{\log 2} \cdot \log n$$

$$Time = \log n$$

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

Time Complexity - O(1), O(log n), O(n), O(nlog n), O(n^2), O(n^3), O(2^n)

modification - '+' / '-' -> time complexity will be in terms n
modification - '*' / '/' -> time complexity will be in terms log n

i = n = 9 ; i > 0 ; i = i / 2 → i = 9, 4, 2, 1, 0 → log n

i = 1 ; i < 9 ; i = i * 2 → i = 1, 2, 4, 8, 16 → log n

```
for(i=0; i<n ; i++)
for(i=n; i> 0 ; i--)
for(i=0; i<n ; i= i+2)
for( i=0 ; i<n ; i= i+100)
```

} T(n) = O(n)

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

Time ∝ 2n

T(n) = O(n)

```
for(i=0; i<n ; i++)   } → n^2
    for(j=0; j<n ; j++)
    { }
for(i=0; j<n ; j++)   → n
```

Time ∝ $n^2 + n$

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

Space Measurement / Space Complexity

- it is approximate measure of space required to execute the algorithm
- Total space is addition of input space and auxillary space

input space - actual space of memory to store input (data)
auxillary space - space of memory which is required to process actual input

- to store 1 value, will need 1 unit of space

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

Input space - n units
Auxillary space - 3 units

Total space = Input space + Auxillary space
= n + 3
= n
S(n) = O(n)

Space Complexity - O(1), O(n), O(n^2), O(n^3).....

Searching Algorithms

- finding the key from collection of values
- There are two types of search
    1. linear search
    2. binary search

Linear search

1. Best case	- key is found at initial places	- O(1)
2. Average case	- key is found at middle places	- O(n)
3. Worst case	- key is found at last places	
	- key is not found	- O(n)

Binary search

1. Best case	- key is found at initial levels	- O(1)
2. Average case	- key is found at middle levels	- O(log n)
3. Worst case	- key is found at last levels	
	- key is not found	- O(log n)

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)

Recursion

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

```
5!      = 5 * 4!
        = 5 * 4 * 3!
        = 5 * 4 * 3 * 2!
        = 5 * 4 * 3 * 2 * 1!
        = 5 * 4 * 3 * 2 * 1
if(num == 1)    return 1
```