Data Structures and Algorithms

Data Structures

- organising data into memory for effiecient access
- on organised data which operations can be performed like add, display(traversing), search, sort
- Types of data structures
  - Basic data structures
    - structure/class
    - array
    - stack
    - queue
    - linked list
  - Advanced data structures
    - Tree
    - Graph
  - Linear data structures
    - in which data/elements are orgnised into memory linearly
    - data/elements can be accessed sequentially
    - all basic data structures are linear
  - Non-linear data structures
    - in which data/elements are orgnised into memory heirarchically
    - data/elements can not be accessed sequentially
    - all Advanced data structures are linear
- to acheive
  - Abstraction
    - All data structures are also known ADT (Abstract Data Type)
  - Reusability
    - we can use data structures as per our need
    - we can use data structures to implement another data structuress
    - we can use data structures in few algorithms
  - Efficiency
    - time - time required to execute any algorithm
    - space - space required in memory to execute any algorithm

Algorithms

- Function - set of instructions to processor/CPU/machine

- Algorithm - set of instructions to human/programmer/developer
- step by step solution to given problem statement
- algorithms always implemented in human languages (English, regional lang)
- eg. Find sum of array elements
  - step 1 - create varaiable to store sum. (sum = 0)
  - step 2 - traverse from 0 to N-1 index
  - step 3 - add every element of array into sum
  - step 4 - print/return final result (sum)
- Algorithms are programming language independent
- Algorithms are like templates/blue prints
- eg. searching, sorting

Efficiency measurement / Algorithm Analysis

- There are two measures of efficiency
  - Time
  - space
- is done to choose best algorithm/solution from multiple
- Types of analysis
  - Exact analysis
  - Approximate analysis

Exact analysis

```
* how much exact time and space will be required to execute
    * time - nS, uS, mS, S
    * space - bytes, KB, MB, ......
* this analysis is dependent on few external parameters
* time is dependent on type of machine, number of processes running at that time
etc
* space is dependent on type and size of variables, type of architectrue (machine)
etc
```

Approximate analysis

- approx budget/measure of time and space
- Asymptotic analysis
  - mathematical way to find time and space complexity
  - behaviour of algorithm for different data element and for different sequence of data elements
  - cases of behaviour
    - best case
    - average case

        - worst case
    - 'Big O'/ O() notation is used to denote time and space complexicity

Time Complexity

- approximate time required to execute any algorithm
- no of iterations of loop used in algorithm

Space Complexity

- approximate space required to execute any algorithm
- total space required is addition of input space and auxillary space
    - input space - space required to store actual data
    - auxillary space - space required to process actual data
- eg. Find sum of array elements
    - int arr[n] - input variable --> input space
    - size, sum, i - processing variables --> auxillary space
    - input space = n units
    - auxillary space = 3 units
    - Total space = input space + auxillary space
    - Total space = n + 3 units
    - Space requirement is directly proportional to total space
    - Space complexity = O(n)
- most of the times only auxillary space complexity analysis is done
    - Auxillary space complexity = O(1)

Searching Algorithms

- searching is finding some key(value) into collection (group of data/values)
- there are two algorithms for searching
    - Linear search
        - traverse collection(array) from one end to another
        - compare each element of array with key
            - if both are equal, then stop
            - if both are not equal, continue
        - Time complexity
            - Best case - O(1)
            - Average case - O(n)
            - Worst case - O(n)
    - Binary search (sorted data)
        - Divide array into two parts
        - compare key with middle element of array
        - if key is matching, return index (mid)
        - if key is less than middle element, search it into left sub array (left partition)

- if key is greater than middle element, search it into right sub array (right partition)
- repeat above stpes till key is not found or will not get invalid partition.
- Time complexity
  - Best case - O(1)
  - Average case - O(log n)
  - Worst case - O(log n)