Sunbeam Institute of Information Technology
Pune and Karad

Data structures and Algorithms

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Course Introduction

- **Course Schedules**
  - 29th Jan 2026 to 20th Feb 2026
  - Mon-Sat: Lecture – 5:00 PM to 8:00 PM

- **Course Format**
  - Participants are encouraged to code alongside (copy code from code-sharing utility in student portal).
  - Post your queries in chat box (on logical end of each topic).
  - Practice assignments will be shared. They are optional. If any doubts, share on WA group (possibly with screenshot). Faculty members or peers can help.

- **Course Goals**
  - Implement each DS & Algorithms from scratch.
  - Understand complexity of algorithms.

- **Programming language**
  - DS & Algorithms are language independent.
  - Classroom coding will be in Java (use IDE of your choice).
  - Will share C++/Python codes at the end of session.

- **Resource sharing**
  - https://github.com/sunbeam-modular/dsa-o-12.git
  - Recorded videos will be available on http://students.sunbeamapps.org

Course Pre-requisites

- **Java**
  - Language Funda
  - Methods
  - Class & Object
  - static members
  - Arrays
  - Collections
- **Python**
  - Language Funda
  - Functions
  - Class & Object
  - Collections
- **C++**
  - Language Funda
  - Functions
  - Class & Object
  - Friend class
  - Arrays
  - Pointers
- **C**
  - Language Funda
  - Functions
  - Structures
  - Arrays
  - Pointers

Course Contents

- **Data Structures:**
  - Linked list
  - Stack
  - Queue
  - Hash Table
  - Binary search tree
  - Heap
  - Graph

- **Algorithms:**
  - Sorting
  - Searching
  - Stack, Queue & Linked list applications
  - Graph algorithms.

- **Algorithm Design Techniques:**
  - Brute Force
  - Divide and Conquer
  - Backtracking
  - Greedy
  - Dynamic Programming

- **Problem solving patterns:**
  - Two/three pointer technique
  - Fast and Slow pointers
  - Sliding window (Fixed / Varibale)
  - Binary search
  - BFS and DFS based
  - Kth element category

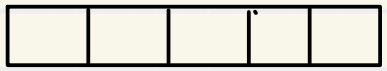
Data Structure

- organising data inside memory for efficient processing.
- operations like add, delete, search, etc which can be performed on data.
- eg stack - push/pop/peek

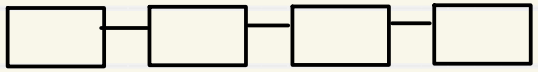
- data structures are used to achieve
  - Abstraction
    - data and organization of it is hidden
    - Abstract Data Types (ADT)
  - Reusability
    - can be used to implement algorithms, can be used to solve problems, to implement other structures
  - Efficiency
    - can be measured in two parameters
      time : time required to execute
      space : space required inside memory

Types of Data structures

Physical data structures



Array



Linked List

Logical data structures

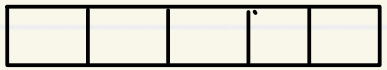
- define relation bet^n data
- are always implemented with the help of physical data structure

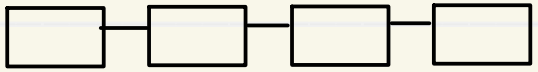
– stack, queue, tree, graph, heap

(Basic)

Linear data structures

- data is organised sequentially/ linearly





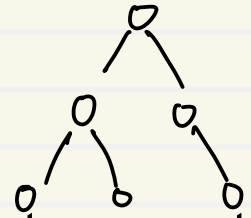
- data can be accessed sequentially

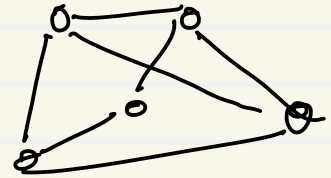
– Array, Linked list, stack, queue
struct / class

(Advanced)

Non linear data structures

- data is organised in multiple levels (hierarchy)



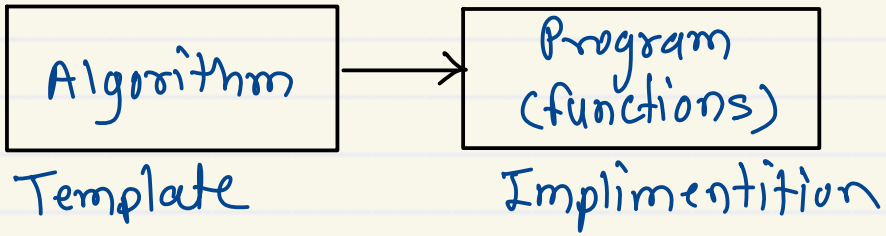


- data can not be accessed sequentially

– tree, graph, heap

Algorithm

- Algorithm is set of instructions given to the programmers
- Algorithm is step by step solution of given problem statement
- As written in human understandable languages, algorithms are programming languages independent
- Algorithms are used as templates and can be implemented in programming languages



Program : set of instructions to machine (CPU)
Algorithm: set of instructions to human(programmers)

Problem - Find sum of array elements

Algorithm :

1. Create space inside memory to store sum and initialize to 0.
2. traverse array from first element to last element (0 to n-1 index)
3. add each element of array into previously calculated sum.
4. print/return final sum.

e.g. searching, sorting

Algorithm analysis

- it is done for efficiency measurement and also known as time/space complexity

- It is done to finding time and space requirements of the algorithm
  1. Time - time required to execute the algorithm (ns , us, ms ..)
  2. Space - space required to execute the algorithm inside memory (bytes .. kb ..)

- finding exact time and space of the algorithm is not possible because it depends on few external factors like
  - time is dependent on type of machine (CPU), number of processes running at that time
  - space is dependent on type of machine (architecture), data types

Algorithm analysis

- Approximate time and space analysis of the algorithm is always done
- Mathematical approach is used to find time and space requirements of the algorithm and it is known as "Asymptotic analysis"
- Asymptotic analysis also tells about behaviour of the algorithm for different input or for change in sequence of input
- This behaviour of the algorithm is observed in three different cases
    1. Best case
    2. Average case
    3. Worst case

- to denote time & space complexity, we use asymptotic notations

Omega (Ω) → lower bound

Big O (O) → Upper bound

Theta (Θ) → Avg / tight bound

Time complexity

- time is directly proportional to number of iterations of the loops used in an algorithm
- To find time complexity/requirement of the algorithm count number of iterations of the loops

1. Find factorial of a number

```
fact = 1;
for( int i = 1; i <= n ; i++)
    fact *= i;
sysout (fact)
```

No. of iterations = n

Time ∝ No. of iterations

Time ∝ n

$$T(n) = O(n)$$

linear growth

2. Print 2D array on console

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

iterations of outer loop = m

iterations of inner loop = n

total iterations = m * n

$m \approx n$

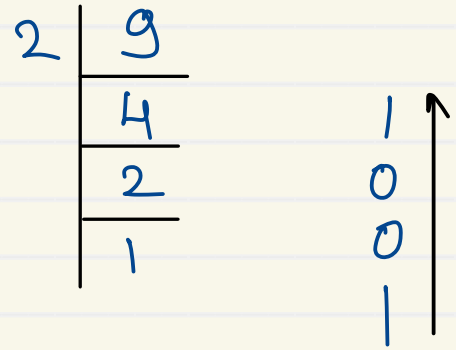
Time ∝ n * n

Time ∝ n^2

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

← quadratic growth

3. Print binary of decimal number



$(9)_{10} = (1001)_2$

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

n	n>0	n%2
9	T	1
4	T	0
2	T	0
1	T	1
0	F	

$n = n, n/2, n/4, n/8 \ldots$

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

$1 = n/2^{itr}$

$2^{itr} = n$

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

$itr \log 2 = \log n$

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

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

$T(n) = O(\log n)$ ← logarithm growth

4. Print table of given number

```
for( i=1 ; i<=10 ; i++)
    sysout( n * i );
```

No. of iterations = 10 (fixed)

↑

do not vary wrt input

- constant time requirement, which is denoted as

$$T(n) = O(1)$$

```
int add ( n1 , n2) {
    return n1 + n2;
}
```

- constant time requirement, which is denoted as

$$T(n) = O(1)$$

Time complexity

Time complexities : O(1), O(log n), O(n), O(n log n), O(n^2), O(n^3), …., O(2^n), ……

Modification : + or - : time complexity is in terms of n
Modification : * or / : time complexity is in terms of log n

```
for(i=0; i<n; i++) → O(n)
for(i=n; i>0; i--) → O(n)
for(i=0; i<n; i+=20) → O(n)
for(i=n; i>0; i/=2) → O(log n)
for(i=1; i<n; i*=2) → O(log n)
```

n = 9, 4, 2, 1
i = 1, 2, 4, 8

```
for(i=1; i<=10; i++) → O(1)
```

```
for(i=0; i<n; i++) → n
  for(j=0; j<n; j++) → n
                          * → O(n^2)
```

```
for(i=0; i<n; i++); → n
for(j=0; j<n; j++); → n
                        + = 2n → O(n)
```

```
for(i=0; i<n; i++) → n
  for(j=n; j>0; j/=2) → logn
                          * → O(n log n)
```

Space complexity

- Finding approximate space requirement of the algorithm to execute inside memory

Total space = Input space + Auxiliary space

Input space: Actual space required to store input

Auxiliary space: Extra space required to process the input

```
int findArraySum(int arr[], int length)
    int sum = 0;
    for( i= 0;  i < length ;  i++)
        sum += arr[i];

    return sum;
}
```

input variable = arr

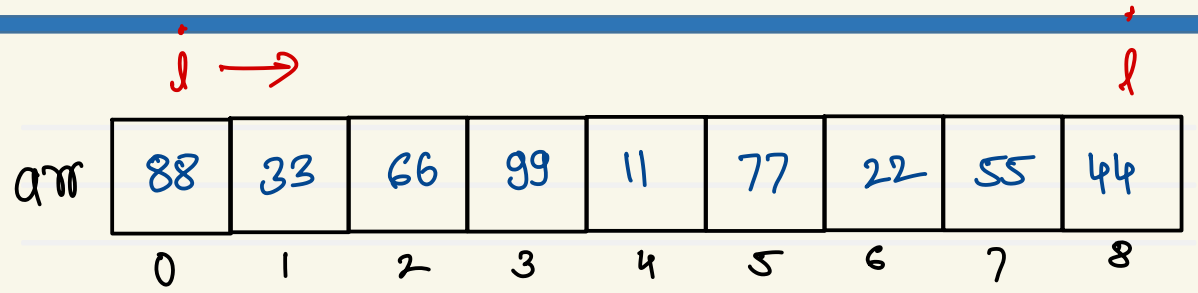
processing variables = length, sum, i

Auxiliary space = 3 units

$$S(n) = O(1)$$

Linear search

1. decide/take key from user
2. traverse collection of data from one end to another
3. compare key with data of collection
   - 3.1 if key is matching
     return index/true
   - 3.2 if key is not matching
     return -1/false



key == arr[i]

77
Key

i = 5 ← key is found

89
Key

i = 9 ← key is not found

Binary search

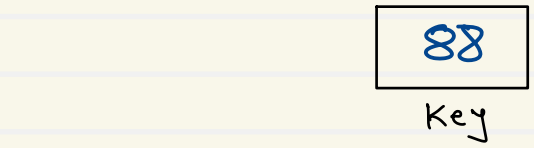
1. take key from user
2. divide array into two parts
   (find middle element)
3. compare middle element with key

3.1 if key is matching
    return index (mid)

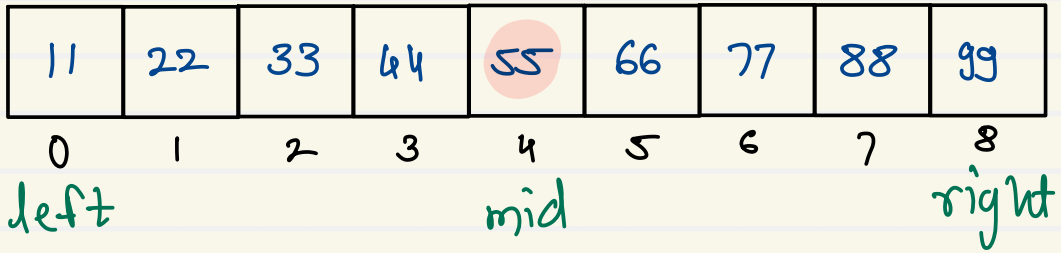
3.2 if key is less than middle element
    search key in left partition

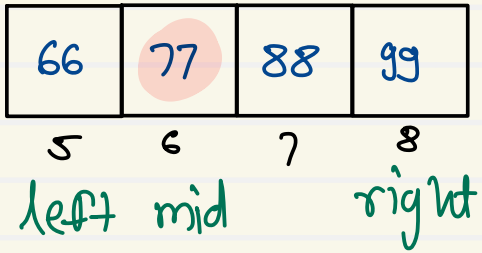
3.3 if key is greater than middle element
    search key in right partition

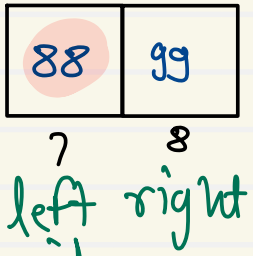
3.4 if key is not matching
    return -1



mid = (left+right) / 2

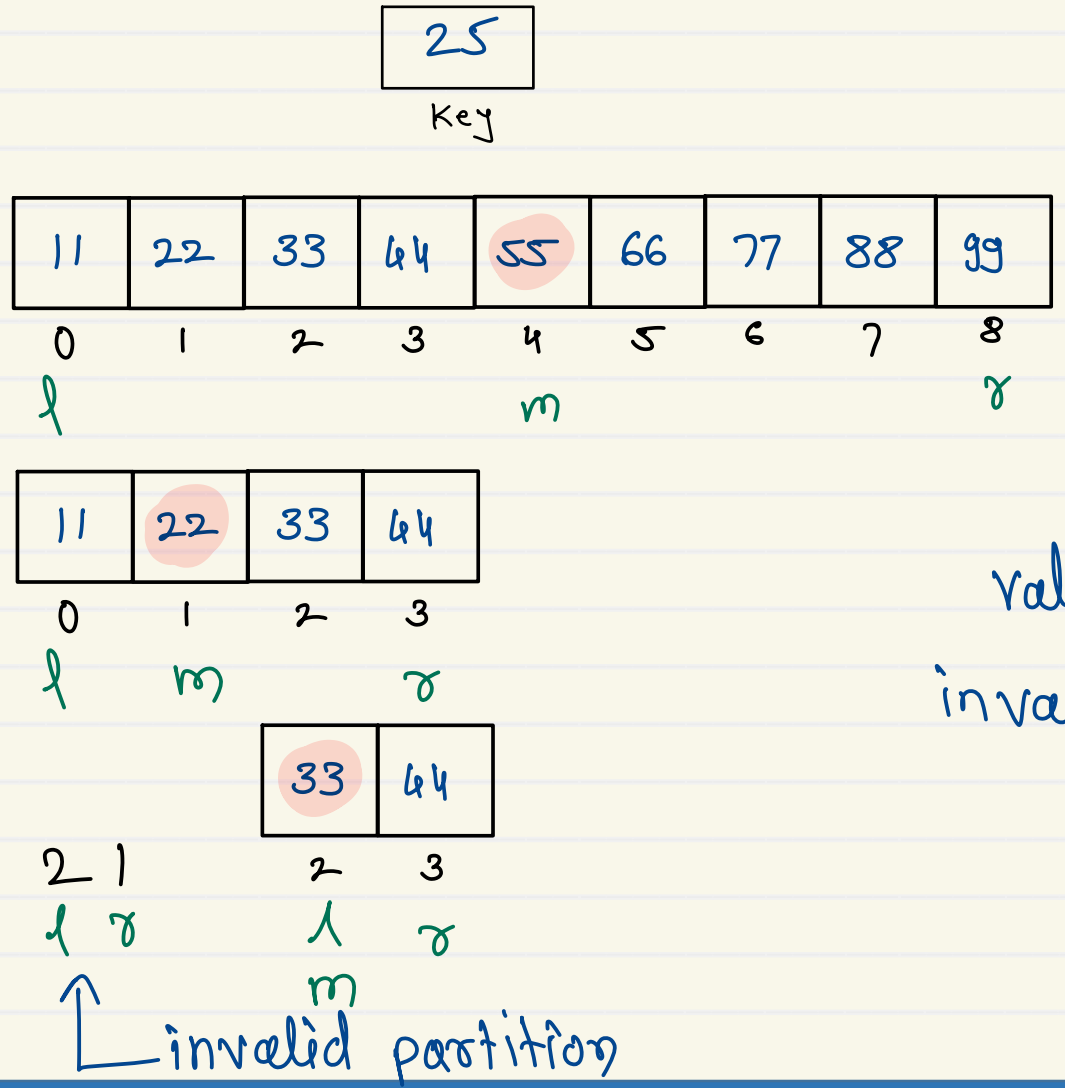






key is found → mid

Binary search



left partition : left → mid-1
right partition : mid+1 → right

valid partition : left <= right
invalid partition : left > right