Introduction to Data Structure

Data Structure using Python(MCAN201)

Table of Content

Data Structure Definition

- Data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.
- A data structure is a special format for organizing and storing data.
- General data structure types include arrays, files, linked lists, stacks, queues, trees, graphs and so on.

Types of data Structure

Depending on the organization of the elements, data structures are classified into two types:

1. Linear data structures: Elements are accessed in a sequential order but it is not compulsory to store all elements sequentially. Examples: Linked Lists, Stacks and Queues.
2. Non – linear data structures: Elements of this data structure are stored/accessed in a non-linear order. Examples: Trees and graphs.

Abstract Data Type

- Abstract Data Types (ADTs) combines the data structures with their operations
- An ADT consists of two parts:

1. Declaration of data
2. Declaration of operations

- Commonly used ADTs include: Linked Lists, Stacks, Queues, Priority Queues, Binary Trees, Dictionaries, Disjoint Sets (Union and Find), Hash Tables, Graphs, and many others.
- Common operations of stack are: creating the stack, pushing an element onto the stack, popping an element from stack, finding the current top of the stack, finding number of elements in the stack, etc.

Data Structure Operations

- Insertion: Operation of storing a new data element in a Data Structure.

- Deletion: The process of removal of data element from a Data Structure.

- Traversal: It involves processing of all data elements present in a Data Structure.

What is an Algorithm?

- An algorithm is the step-by-step unambiguous instructions to solve a given problem.

- Two main criteria for judging the merits of algorithms: correctness (does the algorithm give solution to the problem in a finite number of steps?) and efficiency (how much resources (in terms of memory and time) does it take to execute the).

Why the Analysis of Algorithms?

- Multiple algorithms are available for solving the same problem (for example, a sorting problem has many algorithms, like insertion sort, selection sort, quick sort and many more).
- Algorithm analysis helps us to determine which algorithm is most efficient in terms of time and space consumed.
- The goal of the analysis of algorithms is to compare algorithms (or solutions) mainly in terms of running time but also in terms of other factors (e.g., memory, developer effort, etc.)

Running time Analysis

Running time Analysis determine how processing time increases as the size of the problem (input size) increases.

Input size is the number of elements in the input, and depending on the problem type, the input may be of different types.

The following are the common types of inputs.

- Size of an array
- Polynomial degree
- Number of elements in a matrix
- Number of bits in the binary representation of the input
- Vertices and edges in a graph.

Rate of Growth

- The rate at which the running time increases as a function of input is called rate of growth.

Time Complexity	Name	Example
2^n	Exponential	The tower of Hanoi problem
n^3	Cubic	Matrix multiplication
n^2	Quadratic	shortest path between nodes in a graph
$n \log n$	Linear Logarithmic	sorting n items by divide and conquer -Mergesort
n	Linear	finding an element in an unsorted array
$\log n$	Logarithmic	finding an element in a sorted array

Types of Analysis

- An algorithm can represent with multiple expressions: one for the case where it takes less time and another for the case where it takes more time.
- There are three types of analysis:
- Worst case

➢Defines the input for which the algorithm takes a long time (slowest time to complete).

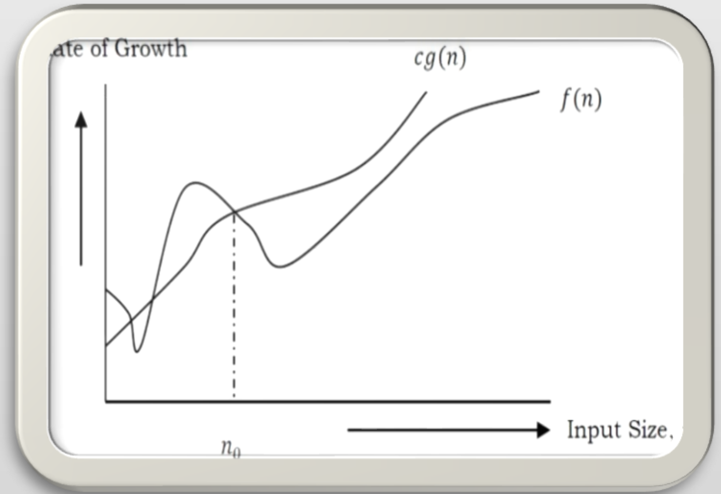
➢Input is the one for which the algorithm runs the slowest.

Types of Analysis

- Best case
  - ➢ Defines the input for which the algorithm takes the least time (fastest time to complete).
  - ➢ Input is the one for which the algorithm runs the fastest.
- Average case
  - ➢ Provides a prediction about the running time of the algorithm.
  - ➢ Run the algorithm many times, using many different inputs that come from some distribution that generates these inputs, compute the total running time (by adding the individual times), and divide by the number of trials.
  - ➢ Assumes that the input is random.

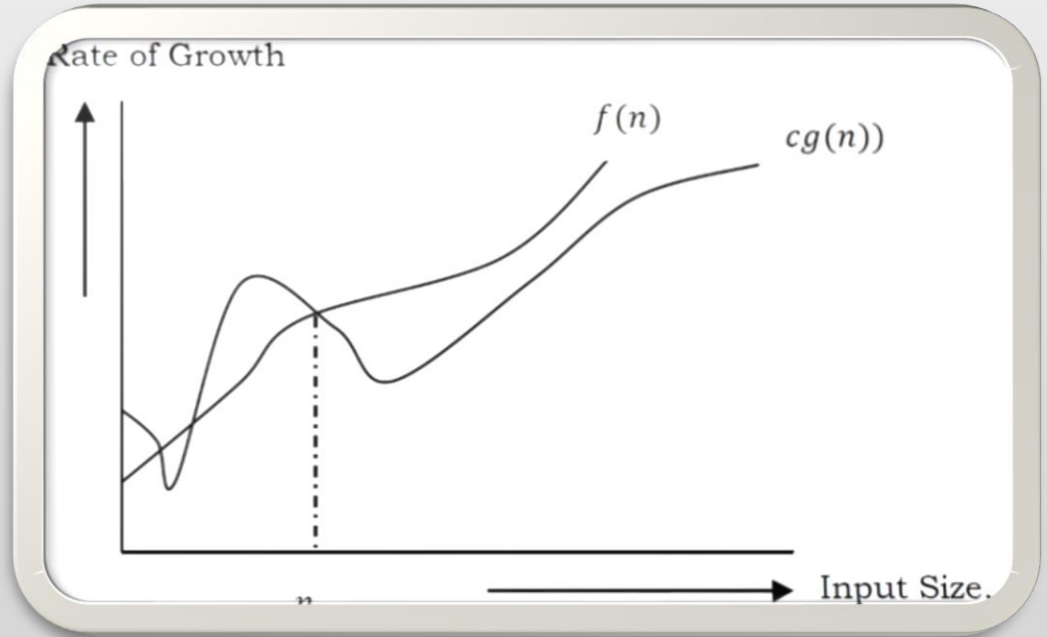
Asymptotic Notations: $Big - O$ Notation

- The upper and lower bounds of time complexity is represented in the form of function $f(n)$.
- This notation gives the tight upper bound of the given function.
- O–notation defined as $O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n > n_0\}$.
- $g(n)$ is an asymptotic tight upper bound for $f(n)$.
- Objective is to give the smallest rate of growth $g(n)$ which is greater than or equal to the given algorithms' rate of growth $f(n)$.

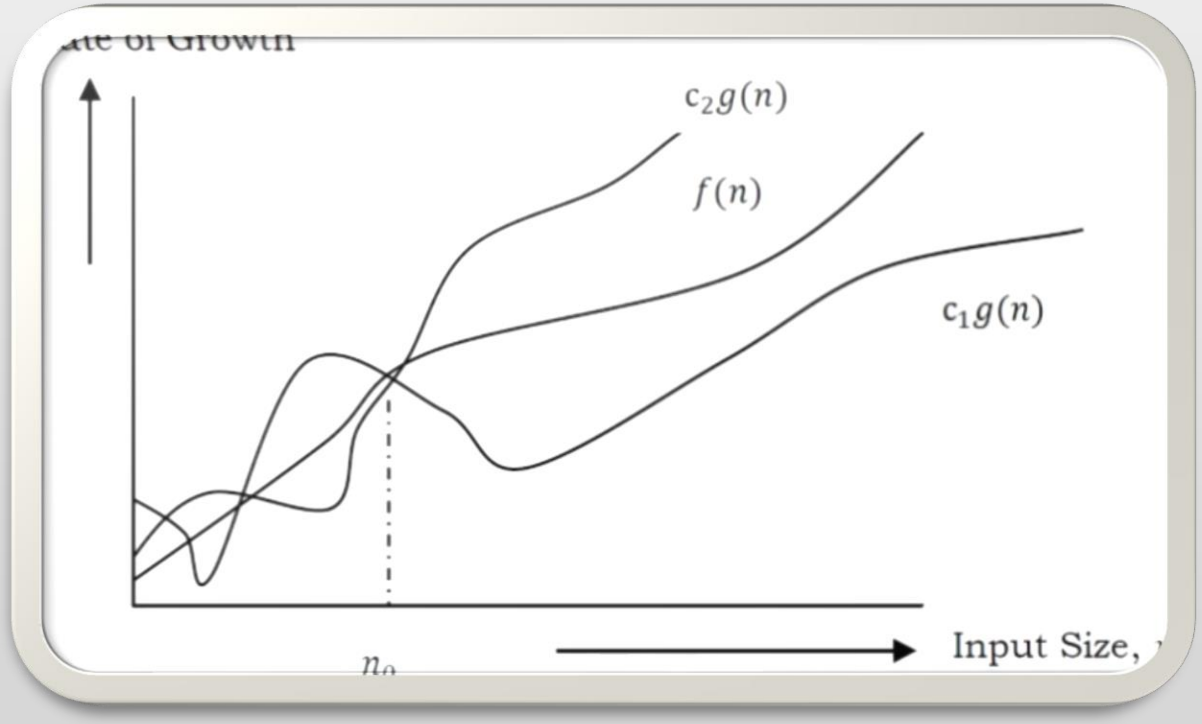


Asymptotic Notations: $Big - \Omega$ Notation

- The Ω notation can be defined as $\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n > n_0\}$.
- $g(n)$ is an asymptotic tight lower bound for $f(n)$.
- Objective is to give the largest rate of growth $g(n)$ which is less than or equal to the given algorithm's rate of growth $f(n)$.



Asymptotic Notations: $Big - \theta$ Notation

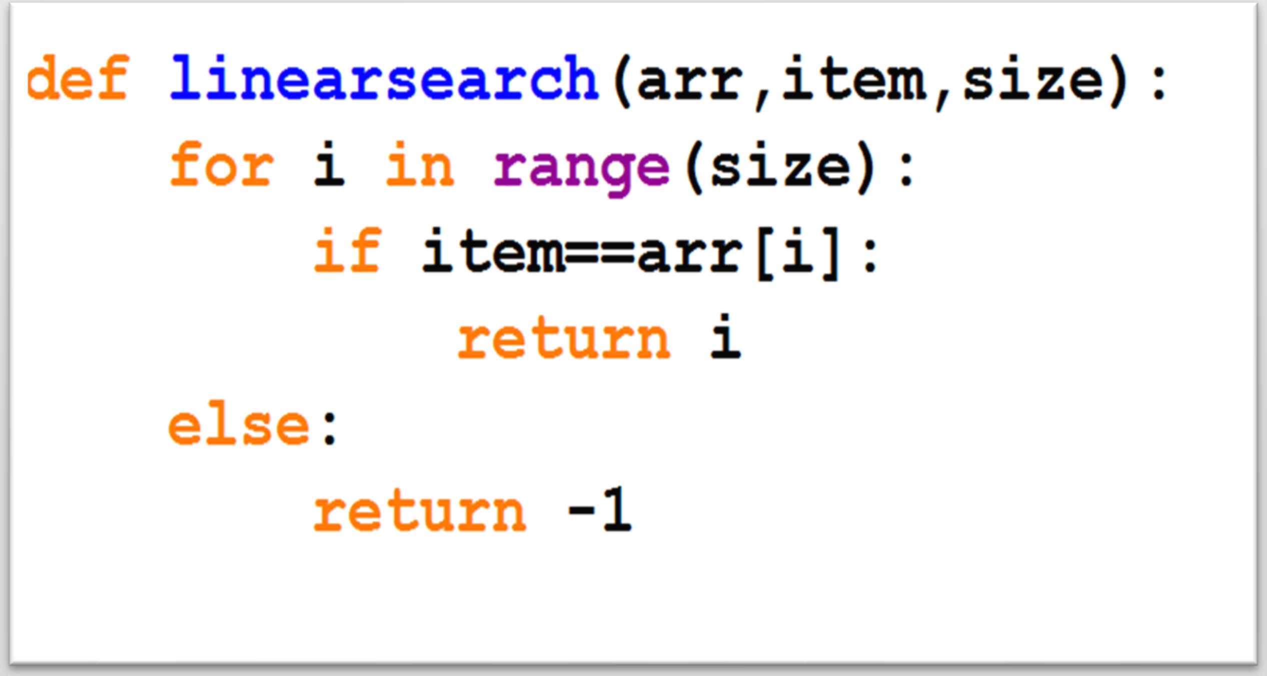


- It is defined as $\Theta(g(n)) = \{f(n)$: there exist positive constants c_1 ,c_2 and n_0 such that $0 \leq c1g(n) \leq f(n) \leq c2g(n)$ for all $n \geq n_0\}$.
- $g(n)$ is an asymptotic tight bound for $f(n)$.
- $\Theta(g(n))$ is the set of functions with the same order of growth as $g(n)$.

- Loops: The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations.
- Nested loops: Analyze from the inside out. Total running time is the product of the sizes of all the loops.
- Consecutive statements: Add the time complexities of each statement.
- If-then-else statements: Worst-case running time: the test, plus either the then part or the else part (whichever is the larger).

Linear Search

- scan the complete array to search for an element and see if the element is there in the given list or not.



```
def linearsearch(arr,item,size):
    for i in range(size):
        if item==arr[i]:
            return i
    else:
        return -1
```

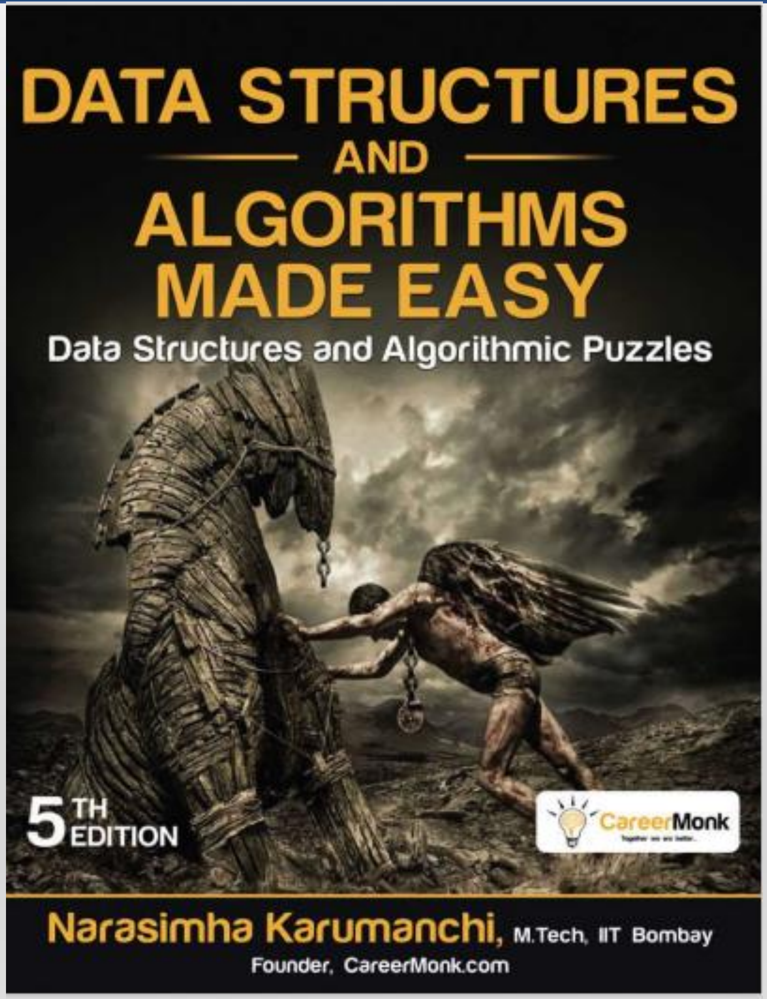
Binary Search

- Binary search is an efficient searching algorithm used in data structures to find the position of a target element in a sorted array or list.
- It works on the divide-and-conquer principle by repeatedly dividing the search interval in half.

Binary Search

```
1.  set $beg = 0$ and $end = len(arr) - 1$
2.  Repeat the step 3 to 9 while $beg \leq end$
3.  $mid = (beg + end)/2$
4.  if $item = arr[mid]$
5.  $return\ mid$
6.  elif $item < arr[mid]$
7.  $end = mid - 1$
8.  else
9.  $beg = mid + 1$
10. else
11. $return\ -1$
```

References


DATA STRUCTURES
AND
ALGORITHMS
MADE EASY
Data Structures and Algorithmic Puzzles
5TH EDITION
CareerMonk
Narasimha Karumanchi, M.Tech, IIT Bombay
Founder, CareerMonk.com

QUIZ