**Index**

[Data Structure 2](file:///D:\B.Tech\%5b%20All%20projects%20%5d\2-%20Git%20not%20posted\Notebook\BCA\DSA\DSA.docx#_Toc171872098)

[What is Data Structure? 3](#_Toc171872099)

[What is ADT in easy language? [Explanation of ABT in easy language] 3](#_Toc171872100)

[What is proper definition of ADTs ? 3](#_Toc171872101)

[Where it is used? 3](#_Toc171872102)

[Algorithm 4](file:///D:\B.Tech\%5b%20All%20projects%20%5d\2-%20Git%20not%20posted\Notebook\BCA\DSA\DSA.docx#_Toc171872103)

[What is Algorithm? 5](#_Toc171872104)

[What is Analysis of Algorithms and what is its purpose? 5](#_Toc171872105)

[What is Running Time Analysis? 5](#_Toc171872106)

[How to Compare Algorithms 5](#_Toc171872107)

[What is Rate of Growth? 5](#_Toc171872108)

[What is Asymptotic Notation? 7](#_Toc171872109)

[Big-O Notation [Upper Bounding Function] 7](#_Toc171872110)

[Omega-Q Notation [Lower Bounding Function] 8](#_Toc171872111)

[Theta-Θ Notation [Order Function] 8](#_Toc171872112)

[What is Recursion? 8](#_Toc171872113)

[Recursion and its Memory structure? 9](#_Toc171872114)

[Diff b/w Recursion Vs Iteration? 9](#_Toc171872115)

[Example of Algo. of Recursion? 10](#_Toc171872116)

[What is Backtracking? 10](#_Toc171872117)

[Example Algorithms of Backtracking 10](#_Toc171872118)

[Sorting Algorithm 11](file:///D:\B.Tech\%5b%20All%20projects%20%5d\2-%20Git%20not%20posted\Notebook\BCA\DSA\DSA.docx#_Toc171872119)

[Some Basic Terminology 12](#_Toc171872120)

[Insertion Sort 12](#_Toc171872121)

**1**

Data Structure Basics

**Objectives**

What is Data Structure?

A data structure is a way to store and organize data in order to facilitate access and modifications. Using the appropriate data structure or structures is an important part of algorithm design. No single data structure works well for all purposes, and so you should know the strengths and limitations of several of them.

Or

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.

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

What is ADT in easy language? [Explanation of ABT in easy language]

Before defining abstract data types, let us consider the different view of system-defined data types. We all know that, by default, all primitive data types (int, float, etc.) support basic operations such as addition and subtraction. The system provides the implementations for the primitive data types. For user-defined data types we also need to define operations. The implementation for these operations can be done when we want to actually use them. That means, in general, user defined data types are defined along with their operations.

What is proper definition of ADTs ?

To simplify the process of solving problems, we combine the data structures with their operations and we call this Abstract Data Types (ADTs). An ADT consists of two parts:

1. Declaration of data

2. Declaration of operations

Where it is used?

Commonly used ADTs include: Linked Lists, Stacks, Queues, Priority Queues, Binary Trees, Dictionaries, Disjoint Sets (Union and Find), Hash Tables, Graphs, and many others.

For example, stack uses LIFO (Last-In-First-Out) mechanism while storing the data in data structures. The last element inserted into the stack is the first element that gets deleted. Common operations of it 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.

**2**

Algorithm Basics

**Objectives**

What is Algorithm?

Algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output in a finite amount of time. An algorithm is thus a sequence of computational steps that transform the input into the output. You can also view an algorithm as a tool for solving a well-specified computational problem.

Or

It is step by step unambiguous instructions to solve a given problem.

there are two main criteria for judging the merits of algorithms:

* 1. **Correctness** (does the algorithm give solution to the problem in a finite number of steps?)
  2. **Efficiency** (how much resources (in terms of memory and time) does it take to execute).

What is Analysis of Algorithms and what is its purpose?

As we have many ways to go from city A to B but we choose that way which suite us the best and is shortest and more reason play role in select which rote to be taken.

Similarly, there are many algorithms for same type of problem like in case of sorting we have many sorting algo. but each one has its own use case.

So, analysis of algo. helps us to find which algo. suits best in which situation.

It also allows us to compare algorithms (or solutions) mainly in terms of running time but also in terms of other factors (e.g., memory, developer effort, etc.)

What is Running Time Analysis?

It is the process of determining how processing time increase as the size of input increases (number of elements to be input).

Also called time complexity.

How to Compare Algorithms

**Execution times:** Not a good measure as execution time is system dependent means if a comp. has i5 & another one has i8 then it is normal that processing speed of i8 is faster than i5 so execution time for comp. with i8 is less than comp. which has i5.

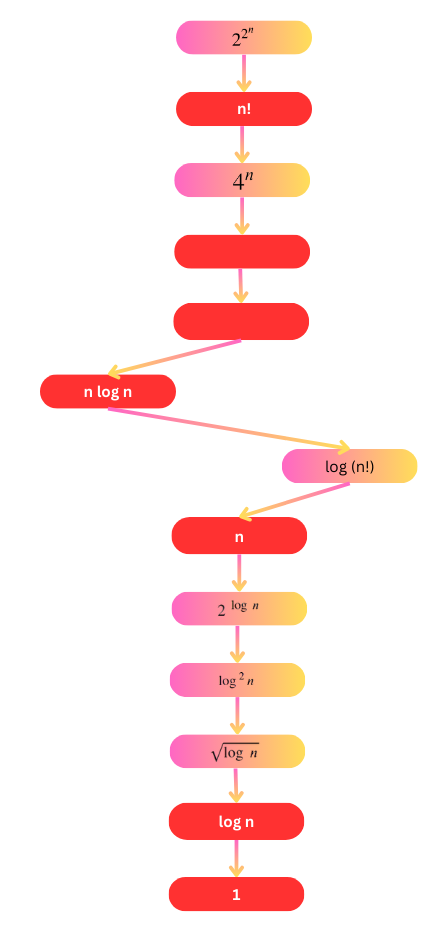
**Number of statements executed:** Not a good measure, since the number of statements varies with the programming language as well as the style of the individual programmer.

**Ideal solution:** Let us assume that we express the running time of a given algorithm as a function of the input size n (i.e., f(n)) and compare these different functions corresponding to running times. This kind of comparison is independent of machine time, programming style, etc.

What is Rate of Growth?

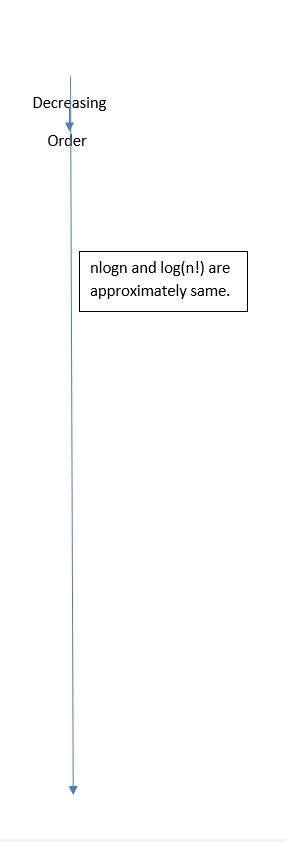
The rate at which the running time increases as a function of input is called rate of growth.

Commonly Used Rates of Growth



**2n**

**n2**



|  |  |  |
| --- | --- | --- |
| Time Complexity | Name | Example |
| 1 | Constant |  |
| Log n | Logarithmic |  |
| n | Linear |  |
| n log n | Linear Logarithmic |  |
| n2 | Quadratic |  |
| n3 | Cubic |  |
| 2n | Exponential |  |

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.

**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.

**Lower Bound <= Avg Time <= Lower Bond**

What is Asymptotic Notation?

They are the notations which allows us to analyse an algorithm running time by identifying the relation b/w it input size and running time.

It includes:

1. Big O Notation (O)
2. Big Omega Notation (Ω)
3. Theta Notation (θ)

Big-O Notation [Upper Bounding Function]

It allows us to find the worst-case value.

f(n) represents our main function.

Big O notation says write f(n) in terms of g(n).

Steps of Big O Notation

**Rules for it 🡪** f(n) = O g(n)

**f(n) <= c. g(n)**

**c > 0**

**n >= k**

**k >= 0**

f(n) = 2n2 + 2 write its Big O notation

Steps

1. f(n) = O (?)
2. 2n2 + 2 <= c. g(?)
3. Remove all constants from main equations
4. So, expression is n2
5. And then decide which is the largest exponent in equation or maximum time taking expression in main equation as here it n2
6. So 2n2 + 2 <= c. g(n2)
7. If we put c = 2, then 2n2 + 2 <= 2. g(n2) which is wrong as the equation is not satisfied.
8. So, range of c 🡪 c >= 3 and for Big O Notation c is equal to least value of range.
9. So, c=3, 2n2 + 2 <= 3. g(n2)
10. 2n2 + 2 <= 3 n2
11. 2 <= n2 , so ±2 <= n but according to condition n < 0 so 2 <= n.
12. So, c=3 for all 2 <= n

Doubt what is use of point 12

Omega-Q Notation [Lower Bounding Function]

Theta-Θ Notation [Order Function]

**Pg 24**

**3**

**Recursion & Backtracking**

**Objectives**

What is Recursion?

Any function which calls itself is called recursive.

A recursive method solves a problem by calling a copy of itself to work on a smaller problem. This is called the **recursion step**. The recursion step can result in many more such recursive calls. **It is important to ensure that the recursion terminates**.

Recursion can do all those tasks which can be done by loops so we have to decide what to use in which situation.

**Use of Recursion**

Recursion is most useful for tasks that can be defined in terms of similar subtasks. Eg: sort, search, and traversal problems often have simple recursive solutions.

A recursive function performs a task in part by calling itself to perform the subtasks. At some point, the function encounters a subtask that it can perform without calling itself. This case, where the function does not recur, is called **the base case**. The base case is the terminating statement. The former, where the function calls itself to perform a subtask, is referred to as **the recursive case**.

Blueprint of Recursion ↴

**function function\_name(parameter) {**

**if(condition){**

**return // Some value/code**

**}**

**else if(condition){**

**return // Some value/code**

**}**

**else{**

**return // Some value/code**

**}**

**}**

any of above can be made base case and other will be Recursion cases

**// Factorial**

**function print(n) {**

**if(n==0) {    // base case**

**return 1;**

**}else{             //recursion case**

**return n\*print(n-1);**

**}**

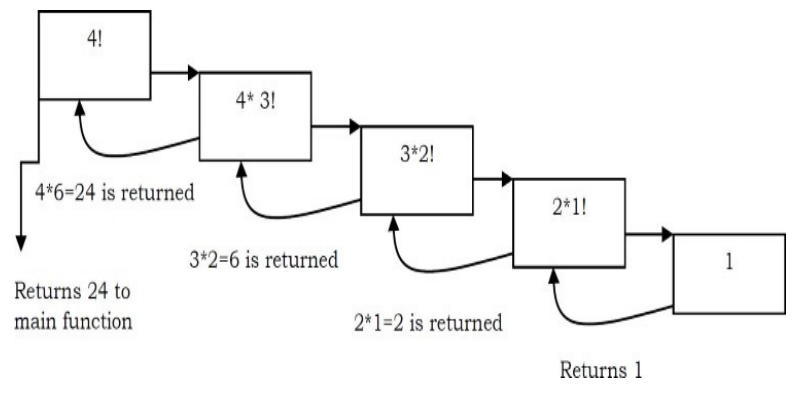
**}**

**print(4)**

Recursion and its Memory structure?

Each recursive call makes a new copy of that method (actually only the variables) in memory. Once a method ends (that is, returns some data), the copy of that returning method is removed from memory.

Memory visualization of Factorial Code ↴



Memory in case of recursion is store in form of stack.

Code & Memory in practical form 🡪 [click here](https://youtu.be/2E9ej_9osMQ)

**Note :** There is a limit to number of times a function can do recursion. The number of recursions is different from compiler to compiler, it is called stack limit.

Diff b/w Recursion Vs Iteration?

About Recursion

* Terminates when a base case is reached.
* Each recursive call requires extra space on the stack frame (memory).
* If we get infinite recursion, the program may run out of memory and result in stack overflow.
* Solutions to some problems are easier to formulate recursively.

**About Iteration**

Repeating a block of code. Eg all type of loops 🡪 for, while, do while

* Terminates when a condition is proven to be false.
* Each iteration does not require extra space.
* An infinite loop could loop forever since there is no extra memory being created.
* Iterative solutions to a problem may not always be as obvious as a recursive solution.

Example of Algo. of Recursion?

* Fibonacci Series, Factorial Finding
* Merge Sort, Quick Sort
* Binary Search
* Tree Traversals and many Tree Problems: InOrder, PreOrder PostOrder
* Graph Traversals: DFS [Depth First Search] and BFS [Breadth First Search]
* Dynamic Programming Examples
* Divide and Conquer Algorithms
* Towers of Hanoi
* Backtracking Algorithms [we will discuss in next section]

What is Backtracking?

Example Algorithms of Backtracking

**Pg 67**

**4**

**Stack**

**Objectives**

**5**

**Linked List**

**Objectives**

**1**

**Introduction**

**Objectives**

**1**

**Introduction**

**Objectives**

**1**

**Introduction**

**Objectives**

**3**

Sorting Algorithm

**Objectives**

Some Basic Terminology

The numbers to be sorted are called keys.

Normally when we want to sort numbers, it’s is often because they are the keys associated with other data, which is known as satellite data.

Insertion Sort

It is an efficient algorithm for sorting a small number of elements.

**Concept:** Think we have card from 1-20 but it is shuffled and now we have to sort them in ascending order. So, we pick upper most card, let it be 10 then another card has to be picked if it is greater then we put it backside of card 10, let it be 12 then another card if it is smaller than 10 then it will be placed above the card no. 10 and is card no. is 11 then it is place b/w 10 & 12 similarly all the cards will be sorted.

Sorting can be in descending order then only we make some changes like card 12 is placed above the card 10 not below it similarly for other.

**Extra**

1. Important Notes
2. Why is it called Asymptotic Analysis?
3. Guidelines for Asymptotic Analysis
4. Simplifying properties of asymptotic notations
5. Commonly used Logarithms and Summations
6. Master Theorem for Divide and Conquer Recurrences
7. Divide and Conquer Master Theorem: Problems & Solutions
8. Master Theorem for Subtract and Conquer Recurrences
9. Variant of Subtraction and Conquer Master Theorem
10. Method of Guessing and Confirming
11. Amortized Analysis
12. Algorithms Analysis: Problems & Solutions
13. Recursion: Problems & Solutions
14. Backtracking: Problems & Solutions

Linked Lists

1. What is a Linked List?
2. Linked Lists ADT
3. Why Linked Lists?
4. Arrays Overview
5. Comparison of Linked Lists with Arrays & Dynamic Arrays
6. Singly Linked Lists
7. Doubly Linked Lists
8. Circular Linked Lists
9. A Memory-efficient Doubly Linked List
10. Unrolled Linked Lists
11. Skip Lists
12. Linked Lists: Problems & Solutions

Stacks

1. What is a Stack?
2. How Stacks are used
3. Stack ADT
4. Applications
5. Implementation
6. Comparison of Implementations
7. Stacks: Problems & Solutions

Queues

1. What is a Queue?
2. How are Queues Used?
3. Queue ADT
4. Exceptions
5. Applications
6. Implementation
7. Queues: Problems & Solutions

Trees

1. What is a Tree?
2. Glossary
3. Binary Trees
4. Types of Binary Trees
5. Properties of Binary Trees
6. Binary Tree Traversals
7. Generic Trees (N-ary Trees)
8. Threaded Binary Tree Traversals (Stack or Queue-less Traversals)
9. Expression Trees
10. XOR Trees
11. Binary Search Trees (BSTs)
12. Balanced Binary Search Trees
13. AVL(Adelson-Velskii and Landis) Trees
14. Other Variations on Trees

Priority Queues and Heaps

1. What is a Priority Queue?
2. Priority Queue ADT
3. Priority Queue Applications
4. Priority Queue Implementations
5. Heaps and Binary Heaps
6. Binary Heaps
7. Heapsort
8. Priority Queues [Heaps]: Problems & Solutions
9. Disjoint Sets ADT
10. Introduction
11. Equivalence Relations and Equivalence Classes
12. Disjoint Sets ADT
13. Applications
14. Tradeoffs in Implementing Disjoint Sets ADT
15. Fast UNION Implementation (Slow FIND)
16. Fast UNION Implementations (Quick FIND)
17. Summary
18. Disjoint Sets: Problems & Solutions

Graph Algorithms

1. Introduction
2. Glossary
3. Applications of Graphs
4. Graph Representation
5. Graph Traversals
6. Topological Sort
7. Shortest Path Algorithms
8. Minimal Spanning Tree
9. Graph Algorithms: Problems & Solutions

Sorting

1. What is Sorting?
2. Why is Sorting Necessary?
3. Classification of Sorting Algorithms
4. Other Classifications
5. Bubble Sort
6. Selection Sort
7. Insertion Sort
8. Shell Sort
9. Merge Sort
10. Heap Sort
11. Quick Sort
12. Tree Sort
13. Comparison of Sorting Algorithms
14. Linear Sorting Algorithms
15. Counting Sort
16. Bucket Sort (or Bin Sort)
17. Radix Sort
18. Topological Sort
19. External Sorting
20. Sorting: Problems & Solutions

Searching

1. What is Searching?
2. Why do we need Searching?
3. Types of Searching
4. Unordered Linear Search
5. Sorted/Ordered Linear Search
6. Binary Search
7. Interpolation Search
8. Comparing Basic Searching Algorithms
9. Symbol Tables and Hashing
10. String Searching Algorithms
11. Searching: Problems & Solutions

Selection Algorithms [Medians]

1. What are Selection Algorithms?
2. Selection by Sorting
3. Partition-based Selection Algorithm
4. Linear Selection Algorithm - Median of Medians Algorithm
5. Finding the K Smallest Elements in Sorted Order
6. Selection Algorithms: Problems & Solutions

Symbol Tables

1. Introduction
2. What are Symbol Tables?
3. Symbol Table Implementations
4. Comparison Table of Symbols for Implementations

Hashing

1. What is Hashing?
2. Why Hashing?
3. HashTable ADT
4. Understanding Hashing
5. Components of Hashing
6. Hash Table
7. Hash Function
8. Load Factor
9. Collisions
10. Collision Resolution Techniques
11. Separate Chaining
12. Open Addressing
13. Comparison of Collision Resolution Techniques
14. How Hashing Gets O(1) Complexity?
15. Hashing Techniques
16. Problems for which Hash Tables are not suitable
17. Bloom Filters
18. Hashing: Problems & Solutions

String Algorithms

1. Introduction
2. String Matching Algorithms
3. Brute Force Method
4. Rabin-Karp String Matching Algorithm
5. String Matching with Finite Automata
6. KMP Algorithm
7. Boyer-Moore Algorithm
8. Data Structures for Storing Strings
9. Hash Tables for Strings
10. Binary Search Trees for Strings
11. Tries
12. Ternary Search Trees
13. Comparing BSTs, Tries and TSTs
14. Suffix Trees
15. String Algorithms: Problems & Solutions

Algorithms Design Techniques

1. Introduction
2. Classification
3. Classification by Implementation Method
4. Classification by Design Method
5. Other Classifications

Greedy Algorithms

1. Introduction
2. Greedy Strategy
3. Elements of Greedy Algorithms
4. Does Greedy Always Work?
5. Advantages and Disadvantages of Greedy Method
6. Greedy Applications
7. Understanding Greedy Technique
8. Greedy Algorithms: Problems & Solutions

Divide and Conquer Algorithms

1. Introduction
2. What is the Divide and Conquer Strategy?
3. Does Divide and Conquer Always Work?
4. Divide and Conquer Visualization
5. Understanding Divide and Conquer
6. Advantages of Divide and Conquer
7. Disadvantages of Divide and Conquer
8. Master Theorem
9. Divide and Conquer Applications
10. Divide and Conquer: Problems & Solutions

Dynamic Programming

1. Introduction
2. What is Dynamic Programming Strategy?
3. Properties of Dynamic Programming Strategy
4. Can Dynamic Programming Solve All Problems?
5. Dynamic Programming Approaches
6. Examples of Dynamic Programming Algorithms
7. Understanding Dynamic Programming
8. Longest Common Subsequence
9. Dynamic Programming: Problems & Solutions

Complexity Classes

1. Introduction
2. Polynomial/Exponential Time
3. What is a Decision Problem?
4. Decision Procedure
5. What is a Complexity Class?
6. Types of Complexity Classes
7. Reductions
8. Complexity Classes: Problems & Solutions

Miscellaneous Concepts

1. Introduction
2. Hacks on Bit-wise Programming
3. Other Programming Questions