DSA Tutorial - Learn Data Structures and Algorithms

Why to Learn DSA?

- Learning DSA boosts your problem-solving abilities and make you a stronger programmer.
- DSA is foundation for almost every software like GPS, Search Engines, AI ChatBots, Gaming Apps, Databases, Web Applications, etc
- Top Companies like **Google, Microsoft, Amazon, Apple, Meta** and many other heavily focus on DSA **i**n interviews.

Are you looking to prepare in limited time ? *Try our free course GfG 160 where we have 160 most asked problems along with well written editorials and video explanations. The course also has 90 bonus problems.*
Do you wish to learn in a scheduled manner ? *Try our ongoing free course DSA Skillup with weekly topic coverage with mock contests, short notes, daily problems and quizzes.*

How to learn DSA?

1. Learn at-least one programming language (C++, Java, Python or JavaScript) and build your basic logic.
2. Learn about Time and Space complexities
3. Learn Data Structures (Arrays, Linked List, etc) and Algorithms (Searching, Sorting, etc).
4. Once you learn main topics, it is important to solve coding problems against some predefined test cases,
5. Solve problems daily using GfG Problem of the Day
6.

1. Logic Building

Once you have learned basics of a programming language, it is recommended that you learn basic logic building

- Logic Building Guide
- Quiz on Logic Building

2. Learn about Complexities

To analyze algorithms, we mainly measure order of growth of time or space taken in terms of input size. We do this in the worst case scenario in most of the cases. Please refer the below links for a clear understanding of these concepts.

- Complexity Analysis Guide
- Quiz on Complexity Analysis

3. Array

Array is a linear data structure where elements are allocated **contiguous memory**, allowing for **constant-time access**.

- Array Guide
- Quiz on Arrays

4. Searching Algorithms

Searching algorithms are used to locate specific data within a large set of data. It helps **find a target value** within the data. There are various types of searching algorithms, each with its own approach and efficiency.

- Searching Guide
- Quiz on Searching

5. Sorting Algorithm

Sorting algorithms are used to **arrange** the elements of a list in a **specific order**, such as numerical or alphabetical. It organizes the items in a systematic way, making it easier to search for and access specific elements.

- Sorting Guide
- Quiz on Sorting

6. Hashing

Hashing is a technique that generates a fixed-size output (hash value) from an input of variable size using mathematical formulas called hash functions. Hashing is commonly used in data structures for efficient searching, insertion and deletion.

- Hashing Guide
- Quiz on Hashing

7. Two Pointer Technique

In Two Pointer Technique, we typically use two index variables from two corners of an array. We use the two pointer technique for searching a required point or value in an array.

- Two Pointer Technique
- Quiz on Two Pointer Technique

8. Window Sliding Technique

In Window Sliding Technique, we use the result of previous subarray to quickly compute the result of current.

- Window Sliding Technique
- Quiz on Sliding Window

9. Prefix Sum Technique

In Prefix Sum Technique, we compute prefix sums of an array to quickly find results for a subarray.

- Prefix Sum Technique
- Quiz on Prefix Sum

10. String

String is a sequence of characters, typically immutable and have limited set of elements (lower case or all English alphabets).

- Strings Guide
- Quiz on Strings

11. Recursion

Recursion is a programming technique where a function **calls itself** within its own definition. It is usually used to solve problems that can be broken down into smaller instances of the same problem.

- Recursion Guide
- Quiz on Recursion

12. Matrix/Grid

Matrix is a two-dimensional array of elements, arranged in **rows** and **columns**. It is represented as a rectangular grid, with each element at the intersection of a row and column.

- Matrix Guide
- Quiz on Matrix/Grid.

13. Linked List

Linked list is a linear data structure that stores data in nodes, which are connected by pointers. Unlike arrays, nodes of linked lists are not stored in contiguous memory locations and can only be **accessed sequentially**, starting from the head of list.

- Linked List Guide
- Quiz on Linked List

14. Stack

Stack is a linear data structure that follows the **Last In, First Out (LIFO)** principle. Stacks play an important role in managing function calls, memory, and are widely used in algorithms like stock span problem, next greater element and largest area in a histogram.

- Stack Guide
- Quiz on Stack

15. Queue

Queue is a linear data structure that follows the **First In, First Out (FIFO)** principle. Queues play an important role in managing tasks or data in order, scheduling and message handling systems.

- Queue Guide
- Quiz on Queue

16. Deque

A **deque** (double-ended queue) is a data structure that allows elements to be added or removed from both ends efficiently.

- Deque Guide
- Quiz on Deque

17. Tree

Tree is a **non-linear, hierarchical** data structure consisting of nodes connected by edges, with a top node called the **root** and nodes having child nodes. It is widely used in **file systems**, **databases**, **decision-making algorithms**, etc.

- Tree Guide
- Quiz on Tree

18. Heap

Heap is a **complete binary tree** data structure that satisfies the **heap property**. Heaps are usually used to implement priority queues, where the **smallest** or **largest** element is always at the root of the tree.

- Heap Guide
- Quiz on Heap

19. Graph

Graph is a **non-linear** data structure consisting of a finite set of **vertices**(or nodes) and a set of **edges**(or links)that connect a pair of nodes. Graphs are widely used to represent relationships between entities.

- Graph Guide
- Quiz on Graph

20. Greedy Algorithm

Greedy Algorithm builds up the solution one piece at a time and chooses the next piece which gives the most obvious and immediate benefit i.e., which is the most **optimal choice at that moment**. So the problems where choosing **locally optimal** also leads to the global solutions are best fit for Greedy.

- Greedy Algorithms Guide
- Quiz on Greedy

21. Dynamic Programming

Dynamic Programming is a method used to solve complex problems by breaking them down into simpler **subproblems**. By solving each subproblem only **once** and **storing the results**, it avoids redundant computations, leading to more **efficient solutions** for a wide range of problems.

- Dynamic Programming Guide
- Quiz on DP

22. Advanced Data Structure and Algorithms

Advanced Data Structures like **Trie**, **Segment Tree**, **Red-Black Tree** and **Binary Indexed Tree** offer significant performance improvements for specific problem domains. They provide efficient solutions for tasks like fast prefix searches, range queries, dynamic updates, and maintaining balanced data structures, which are crucial for handling large datasets and real-time processing.

- Trie
- Segment Tree
- Red-Black Tree
- Binary Indexed Tree
- Practice Advanced Data Structures

23. Other Algorithms

Bitwise Algorithms: Operate on individual bits of numbers.

- Bitwise Algorithms Guide
- Quiz on Bit Magic

Backtracking Algorithm : Follow Recursion with the option to **revert and traces back** if the solution from current point is not feasible.

- Backtracking Guide
- Quiz on Backtracking

Divide and conquer: A strategy to solve problems by dividing them into **smaller subproblems**, solving those subproblems, and combining the solutions to obtain the final solution.

- Divide and Conquer Guide
- Quiz on Divide and Conquer

Branch and Bound : Used in combinatorial optimization problems to systematically search for the best solution. It works by dividing the problem into smaller subproblems, or branches, and then eliminating certain branches based on bounds on the optimal solution. This process continues until the best solution is found or all branches have been explored.

- Branch and Bound Algorithm

Geometric algorithms are a set of algorithms that solve problems related to **shapes**, **points**, **lines** and polygons.

- Geometric Algorithms
- Practice Geometric Algorithms

Randomized algorithms are algorithms that use **randomness** to solve problems. They make use of random input to achieve their goals, often leading to **simpler** and more **efficient solutions**. These algorithms may **not product same result** but are particularly useful in situations when a **probabilistic approach** is acceptable.

- Randomized Algorithms

Hoping you have learned a programming language of your choice, here comes the next stage of the roadmap - Learn about Time and Space Complexities.