[Skip to content](https://chatgpt.com/c/6867ec48-e218-8001-896e-fcb1e4b185e4#main)

**Chat history**

**You said:**

I will give you the content as I'm making notes of basics for DSA for MAANG.

**ChatGPT said:**

Perfect, Manoj! Go ahead and share the content you're making notes of for DSA. I’ll review it for:

* **Clarity and correctness**
* **Relevance for MAANG-level interviews**
* **Suggestions for improvement (if any)**
* **Adding examples or diagrams if needed**

Once you send it, I can also help you organize it into an easy-to-revise format (PDF, Notion-style, or anything you prefer). Ready when you are.

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... 17 M DSA Course DSA Tutorial Data Structures Algorithms Array Strings Linked List Stack Queue Tree Graph Searching Sorting Recursion Dynamic Programming Binary Tree Binary Search Tree Heap Hashing Divide & Conquer Mathematical Geometric Bitwise Greedy Backtracking Branch and Bound Matrix Pattern Searching Randomized Explore GfG Courses Share Your Experiences IBM Placement Paper | Logical Reasoning Set - 5 I Can't Use Logic In Programming. What Should I Do? HCL Placement Paper | Logical Reasoning Set - 5 IBM Placement Paper | Logical Reasoning Set 3 HCL Placement Paper | Logical Reasoning Set - 3 DSA to DevelopmentCourse Logic Building Problems Last Updated : 09 Jun, 2025 Logic building is about creating clear, step-by-step methods to solve problems using simple rules and principles. It’s the heart of coding, enabling programmers to think, reason, and arrive at smart solutions just like we do. Here are some tips for improving your programming logic: Understand the problem: Read and understand the problem statement. Generate Examples: Create additional input and output cases for each problem. Draw observations: Draw observations and patterns based on the examples you created. Start with Basic: First, think of the basic way to solve the problem then optimize the solution. Basic Problems Check Even or Odd Multiplication Table Sum of Naturals Sum of Squares of Naturals Swap Two Numbers Closest Number Dice Problem Nth Term of AP Easy Problems Sum of Digits Reverse Digits Prime Testing Check Power Distance between Two Points Valid Triangle Overlapping Rectangles Factorial of a Number Pair Cube Count GCD or HCF LCM of Two Numbers Perfect Number Add Two Fraction Day of the Week Nth Fibonacci Number Decimal to Binary N-th term of 1, 3, 6, 10, 15, 21… Armstrong Number Palindrome Number Digit Root Medium Problems Square Root 3 Divisors Divisible by 4 Divisibility by 11 Divisibility by 13 K-th Digit in a^b Fraction to Recurring Decimal Recurring Sequence in a Fraction Compute nPr Compute nCr Pascal’s Triangle All Factor (Or Divisors) Prime Factorization Largest Prime factor Modular Exponentiation nth Catalan Number Binomial Coefficient Power Set Next Permutation Hard Problems Sieve of Eratosthenes Clock Angle Problem Tower of Hanoi Rat and Poisoned 8 puzzle Problem Determinant of a Matrix Euler's Totient Function Josephus Problem Recommended Links Top Logic Building Interview Problems Practice Logic Building Problems Pattern Printing Problems Mathematical Algorithms Comment More info Campus Training Program Next Article Logic Building Problems course-img 57k+ interested Geeks Data Science Training Program Explore course-img 2k+ interested Geeks System Design Training Program Explore course-img 136k+ interested Geeks Mastering Generative AI and ChatGPT Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox -> Now make the simple notes about it and also highlight the standard formulas

**ChatGPT said:**

Great, Manoj! You've shared the **logic building problem list** from GeeksforGeeks, which is a great way to build strong fundamentals before deep-diving into DSA for MAANG-level interviews.

Here's a **clean and organized version of your notes** with:

* 📘 Categorized problems (Basic → Hard)
* 🧠 Concepts being tested
* ✅ Key formulas or concepts highlighted in **bold**
* 🧮 Short notes added wherever useful

**📚 Logic Building Problems – DSA Foundation Notes (MAANG Prep)**

**🟢 Basic Problems**

| **Problem** | **Concept** |
| --- | --- |
| Check Even or Odd | if (num % 2 == 0) → Even |
| Multiplication Table | Use loop: for i in range(1,11) |
| Sum of Naturals | **Formula:** n\*(n+1)/2 |
| Sum of Squares of Naturals | **Formula:** n\*(n+1)\*(2n+1)/6 |
| Swap Two Numbers | Temp var or arithmetic swap: a = a + b; b = a - b; a = a - b; |
| Closest Number | Round or compare with floor/ceil |
| Dice Problem | Probabilities or combinations |
| Nth Term of AP | **Formula:** a + (n-1)d |

**🟡 Easy Problems**

| **Problem** | **Concept** |
| --- | --- |
| Sum of Digits | Loop + num % 10 |
| Reverse Digits | Multiply rev = rev\*10 + num%10 |
| Prime Testing | Check divisibility from 2 to √n |
| Check Power | Use log or divide repeatedly |
| Distance between 2 Points | **Formula:** √((x2-x1)² + (y2-y1)²) |
| Valid Triangle | **Rule:** a + b > c && a + c > b && b + c > a |
| Overlapping Rectangles | Compare coordinates |
| Factorial | n! = n × (n-1)! |
| Pair Cube Count | Brute-force + set check |
| GCD or HCF | **Euclidean Algorithm:** gcd(a,b) = gcd(b, a%b) |
| LCM of Two Numbers | **LCM(a,b) = (a\*b)/GCD(a,b)** |
| Perfect Number | Sum of divisors = number |
| Add Two Fractions | a/b + c/d = (ad + bc)/bd then reduce |
| Day of the Week | Use Zeller’s Congruence or datetime lib |
| Nth Fibonacci Number | Use loop or DP |
| Decimal to Binary | Divide by 2, reverse remainders |
| N-th Term (1,3,6,10…) | **Formula:** n(n+1)/2 |
| Armstrong Number | Sum of power of digits = number |
| Palindrome Number | Same forward and reverse |
| Digit Root | Sum digits until single digit |

**🟠 Medium Problems**

| **Problem** | **Concept** |
| --- | --- |
| Square Root | Binary Search or sqrt() |
| 3 Divisors | Primes whose squares ≤ n |
| Divisible by 4/11/13 | Use digit rules or modular check |
| K-th Digit in a^b | Use pow(a, b) & string indexing |
| Fraction to Recurring Decimal | Long division with hashmap |
| Recurring Sequence in Fraction | Same logic as above |
| Compute nPr | **Formula:** n! / (n-r)! |
| Compute nCr | **Formula:** n! / (r! \* (n-r)!) |
| Pascal’s Triangle | Each term: C(n,k) |
| All Factors of n | Loop till √n |
| Prime Factorization | Divide by smallest primes |
| Largest Prime Factor | Find max in factor list |
| Modular Exponentiation | (a^b) % m with binary exponentiation |
| nth Catalan Number | **Formula:** C(2n, n) / (n + 1) |
| Binomial Coefficient | Same as nCr |
| Power Set | 2^n subsets |
| Next Permutation | STL: next\_permutation() or custom logic |

**🔴 Hard Problems**

| **Problem** | **Concept** |
| --- | --- |
| Sieve of Eratosthenes | **Efficient prime generation** |
| Clock Angle Problem | **Formula:** Hour: H\*30 + M\*0.5 Min: M\*6 Angle = ` |
| Tower of Hanoi | **Recursive relation:** T(n) = 2T(n-1) + 1 |
| Rat and Poisoned Bottle | Binary logic (find poisoned using log₂n) |
| 8 Puzzle Problem | BFS/DFS/Heuristic Search (A\*) |
| Determinant of a Matrix | Recursion + cofactor expansion |
| Euler's Totient Function | Count numbers ≤ n coprime to n |
| Josephus Problem | Recursion: f(n,k) = (f(n-1,k) + k) % n |

**📌 Key Takeaways**

1. Start with **easy problems** to build logical confidence.
2. Focus on formulas & patterns—they’re often reused in **MAANG DSA rounds**.
3. Practice **implementation** of all math-based problems like GCD, LCM, Factorials, Catalan Numbers, and Modular Exponentiation.
4. Understand **binary search-based logic** in square roots and other high-efficiency problems.

Would you like me to:

* Convert this into a **Notion-friendly table format**?
* Export as a **PDF or DOCX** file for your offline use?
* Add **visual flowcharts/diagrams** for problems like Tower of Hanoi or Sieve?

Let me know what format you prefer next.

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... 17 M DSA Course DSA Algorithms Analysis of Algorithms Sorting Searching Greedy Recursion Backtracking Dynamic Programming Divide and Conquer Geometric Algorithms Mathematical Algorithms Pattern Searching Bitwise Algorithms Branch & Bound Randomized Algorithms Explore GfG Courses Share Your Experiences Analysis of Algorithms Complete Guide On Complexity Analysis - Data Structure and Algorithms Tutorial Why is Analysis of Algorithm important? Types of Asymptotic Notations in Complexity Analysis of Algorithms Worst, Average and Best Case Analysis of Algorithms Asymptotic Analysis How to Analyse Loops for Complexity Analysis of Algorithms Sample Practice Problems on Complexity Analysis of Algorithms Basics on Analysis of Algorithms Asymptotic Notations Some Advance Topics Complexity Proofs DSA to DevelopmentCourse Analysis of Algorithms Last Updated : 12 May, 2025 Analysis of Algorithms is a fundamental aspect of computer science that involves evaluating performance of algorithms and programs. Efficiency is measured in terms of time and space. Basics on Analysis of Algorithms: Why is Analysis Important? Order of Growth Asymptotic Analysis Worst, Average and Best Cases of Algorithms Asymptotic Notations: Big-O Notation Theta - Θ Notation Big – Ω (Big- Omega) Notation Time Complexity Space Complexity Big O vs, Big Omega vs Theta Examples of Big-O analysis Practice Questions on Time Complexity Analysis Quiz on Time Complexity Analysis Analysis Examples Analyzing Loops Analyzing Recursive Functions Amortized Analysis Some Advance topics: P, NP, CoNP, NP hard and NP complete Proof that Clique Decision problem is NP-Complete Proof that Independent Set in Graph theory is NP Complete Prove that a problem consisting of Clique and Independent Set is NP Complete Prove that Dense Subgraph is NP Complete by Generalisation Prove that Sparse Graph is NP-Complete Comment More info Campus Training Program Next Article Complete Guide On Complexity Analysis - Data Structure and Algorithms Tutorial course-img 1418k+ interested Geeks Data Structures and Algorithms - Self Paced [Online Course] Explore course-img 112k+ interested Geeks Data Structures & Algorithms in JavaScript - Self Paced Course Explore course-img 344k+ interested Geeks Data Structures & Algorithms in Python - Self Paced Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Thanks, Manoj! You're doing great by collecting foundational content from GeeksforGeeks. Based on what you've shared, here's a **well-organized and simplified set of notes** for your **"Analysis of Algorithms"** section from your DSA prep.

**📘 Analysis of Algorithms – MAANG-Focused DSA Notes**

**🔍 Why is Analysis of Algorithms Important?**

* Helps **measure efficiency** of code.
* Predicts **runtime and memory usage**.
* Ensures **scalability** with large inputs (essential for MAANG coding rounds).
* Aids in choosing the best algorithm among alternatives.

**📈 Order of Growth**

Represents how runtime **grows with input size**n.

Common orders:

* O(1) → Constant
* O(log n) → Logarithmic
* O(n) → Linear
* O(n log n) → Log-linear
* O(n²) → Quadratic
* O(2ⁿ) → Exponential

**⏱️ Asymptotic Analysis**

Used to describe runtime **as input size tends to infinity.**

**✅ Best Case**

* Minimum steps algorithm takes (least input work).
* Rarely used for judging performance.

**✅ Average Case**

* Expected performance over many inputs.
* Useful, but hard to determine for complex algorithms.

**✅ Worst Case**

* Maximum steps in the **most difficult scenario.**
* Most commonly used in interviews.

**📏 Asymptotic Notations**

| **Notation** | **Meaning** |
| --- | --- |
| **Big-O (O)** | **Upper bound** – Worst case |
| **Big-Ω (Omega)** | **Lower bound** – Best case |
| **Θ (Theta)** | **Tight bound** – Average case, exact bound |

**📦 Time Complexity**

Measures how time taken grows with input size.

**🔁 Analyzing Loops**

for (int i = 0; i < n; i++) *// O(n)* for (int i = 0; i < n; i += 2) *// O(n/2) = O(n)* for (int i = 1; i < n; i \*= 2) *// O(log n)*

**🔄 Nested Loops**

for (int i = 0; i < n; i++) { for (int j = 0; j < n; j++) { *// O(n²)* } }

**💾 Space Complexity**

* Refers to **extra memory used** (not input).
* Example:
  + No extra data structures → O(1)
  + Using an array of size n → O(n)

**🧮 Common Big-O Examples**

| **Algorithm** | **Time Complexity** |
| --- | --- |
| Linear Search | O(n) |
| Binary Search | O(log n) |
| Bubble Sort | O(n²) |
| Merge Sort | O(n log n) |
| Quick Sort (avg) | O(n log n) |
| Quick Sort (worst) | O(n²) |

**🧠 Amortized Analysis**

* Analyzes the **average time per operation** over a sequence of operations.
* Example: **Dynamic array resizing** (append can be O(1) *amortized*, even if some resizes take O(n)).

**🔒 Advanced Topics (NP, P, NP-Complete)**

| **Concept** | **Meaning** |
| --- | --- |
| **P** | Problems solvable in polynomial time |
| **NP** | Problems verifiable in polynomial time |
| **NP-Complete** | Hardest problems in NP (if one is solvable in P, all are) |
| **NP-Hard** | At least as hard as NP-Complete; not necessarily in NP |

**📌 Popular Proof Topics (For Theory Interviews):**

* Proving **Clique**, **Independent Set**, **Dense Subgraph**, **Sparse Graph** are NP-Complete.

**🧪 Practice Suggestions**

* Do time complexity analysis of:
  + Recursive Fibonacci
  + Factorial
  + Binary Search
  + Merge Sort
* Write custom recursive functions and try to **analyze them step-by-step**.

**✅ Conclusion**

| **Skill** | **Must-Know For MAANG** |
| --- | --- |
| Big-O, Θ, Ω | ✅ |
| Time vs. Space Tradeoffs | ✅ |
| Loop Analysis | ✅ |
| Recursion Complexity | ✅ |
| Amortized Analysis | ✅ (esp. for dynamic arrays, hashing) |
| NP-Completeness Theory | ✅ (for system design/CS rounds) |

Would you like:

* A combined PDF of this **+ logic-building notes**?
* Flashcards for Big-O concepts and examples?
* A practice set with increasing complexity?

Let me know, I’ll customize accordingly.

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... M DSA Course DSA Interview Questions on Array Practice Array MCQs on Array Tutorial on Array Types of Arrays Array Operations Subarrays, Subsequences, Subsets Reverse Array Static Vs Arrays Array Vs Linked List Array | Range Queries Advantages & Disadvantages ▲ Explore GfG Courses Share Your Experiences DSA Tutorial - Learn Data Structures and Algorithms Array Data Structure Guide String in Data Structure Matrix Data Structure Searching Algorithms Sorting Algorithms Hashing in Data Structure Two Pointers Technique Sliding Window Technique Prefix Sum Array - Implementation and Applications Linked List Data Structure Stack Data Structure Queue Data Structure Recursive Algorithms Tree Data Structure Heap Data Structure Greedy Algorithms Dynamic Programming or DP Divide and Conquer Algorithm Graph Algorithms Competitive Programming - A Complete Guide DSA to DevelopmentCourse Array Data Structure Guide Last Updated : 13 Apr, 2025 In this article, we introduce array, implementation in different popular languages, its basic operations and commonly seen problems / interview questions. An array stores items (in case of C/C++ and Java Primitive Arrays) or their references (in case of Python, JS, Java Non-Primitive) at contiguous locations. It offers mainly the following advantages over other data structures. Random Access : i-th item can be accessed in O(1) Time as we have the base address and every item or reference is of same size. Cache Friendliness : Since items / references are stored at contiguous locations, we get the advantage of locality of reference. It is not useful in places where we have operations like insert in the middle, delete from middle and search in a unsorted data. It is a fundamental and linear data structure using which we build other data structures like Stack Queue, Deque, Graph, Hash Table, etc. Learn Basics of Array: Introduction to Arrays Applications of Array Array in Different Language: Arrays in C Vector in C++ STL Arrays in Java ArrayList in Java List in Python Arrays in C# Arrays in JavaScript Basic Problems on Array: Print Alternates Linear Search Largest Element Second Largest Largest Three Leaders in an array Check if Sorted Remove Duplicates from Sorted Generate all Subarrays Reverse an Array Rotate an Array Zeroes to End Min Increments to Make Equal Min Cost to Make Size 1 Prerequisite for the Remaining Problems Binary Search Selection Sort, Insertion Sort, Binary Search, QuickSort, MergeSort, CycleSort, and HeapSort Sort in C++ / Sort in Java / Sort in Python / Sort in JavaScript Two Pointers Technique Prefix Sum Technique Basics of Hashing Window Sliding Technique Easy Problems on Array: Distinct Elements Duplicate within K Distance Make Even Positioned Greater Sum of all Subarrays Stock Buy and Sell – Multiple Transactions Single Among Doubles Missing Number Missing and Repeating Only Repeating from 1 to n-1 Sorted Subsequence of Size 3 Binary Sort Stable Binary Sort Stable Binary Sort with O(1) Space Sort in Wave Form Max Subarray Sum K Concatenation Max Subarray Sum Max Subarray Product Equilibrium index Two Sum - Find if there is a Pair Two Sum - Closest Pair [More problems on 2 Sum in Medium Section] Chocolate Distribution Problem Union of two arrays Intersection of two arrays Union of two sorted arrays Intersection of two sorted arrays Medium Problems on Array: Make arr[i] = i Sort an array of 1 to n Reorder according to given indexes Minimum Swaps to Sort Min in a Sorted Rotated Max in a Sorted Rotated Rotation Count in Rotated Sorted Search in a Sorted Rotated Nearly Sorted (or K sorted) Product Except Self K-th Largest Sum Subarray Smallest missing number Smallest subarray with sum greater than x Inversion count Merge Overlapping Intervals Sort an array of 0s, 1s and 2s Merge with O(1) extra space Majority Element Peak element Count possible triangles Sub-array with given sum Longest Subarray with Equal 0s and 1s Longest Common Span in Two Binary Arrays Construct an array from its pair-sum array 2 Sum - All Pairs 2 Sum - Distinct Pairs 3 Sum - Find Any 3 Sum - Closest Triplet 4 Sum - Find Any [More problems on 4 Sum in Hard Section] Hard Problems on Array: Surpasser Count Trapping Rain Water Top K Frequent Elements Kth Missing Positive Number in a Sorted Array Stock Buy and Sell - At Most K Transactions Stock Buy and Sell - At Most 2 Transactions Median in a Stream Smallest Difference Triplet from 3 arrays Max occurred in n ranges 3 Sum - Distinct Triplets 3 Sum - All Triplets 4 Sum - Distinct Quadruples 4 Sum - All Quadruples 4 Sum - Closest Quadruple Expert Problems for Competitive Programmers MO’s Algorithm Square Root (Sqrt) Decomposition Algorithm Sparse Table Range sum query using Sparse Table Range Minimum Query (Square Root Decomposition and Sparse Table) Range LCM Queries Merge Sort Tree for Range Order Statistics Minimum number of jumps to reach end Space optimization using bit manipulations Max value of Sum( i\*arr[i]) with only rotations allowed Quick Links : ‘Practice Problems’ on Arrays Top Array Interview Questions ‘Quizzes’ on Arrays Comment More info Campus Training Program Next Article String in Data Structure course-img 1417k+ interested Geeks Data Structures and Algorithms - Self Paced [Online Course] Explore course-img 344k+ interested Geeks Data Structures & Algorithms in Python - Self Paced Explore course-img 112k+ interested Geeks Data Structures & Algorithms in JavaScript - Self Paced Course Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox Similarly this one

**ChatGPT said:**

Absolutely, Manoj! Here's a clean, organized, and **MAANG-ready version of your notes** for the **Array Data Structure**, categorized and simplified with **concepts, advantages, problems, and formulas** for better understanding and revision.

**📘 DSA Notes – Array Data Structure (MAANG-Focused)**

**🔍 What is an Array?**

* A **linear data structure** that stores elements at **contiguous memory locations**.
* Elements are of the **same type**.
* Allows **random access** using indices in **O(1)** time.

**✅ Advantages of Arrays**

| **Advantage** | **Explanation** |
| --- | --- |
| **Random Access** | arr[i] is accessed in **O(1)** time |
| **Cache Friendly** | Stored in contiguous memory (better performance) |
| **Simple to implement** | Base of many other data structures: Stack, Queue, Matrix, etc. |

**❌ Disadvantages**

* Fixed size (unless using dynamic structures like ArrayList, Vector)
* Insertion/Deletion from middle = **O(n)**
* Searching in unsorted array = **O(n)**

**🛠️ Array Operations**

| **Operation** | **Time Complexity** |
| --- | --- |
| Access arr[i] | O(1) |
| Search (Unsorted) | O(n) |
| Search (Sorted + Binary Search) | O(log n) |
| Insertion (Middle) | O(n) |
| Deletion (Middle) | O(n) |

**🧠 Key Concepts**

* **Subarray**: Continuous part of the array.
* **Subsequence**: Not necessarily contiguous, maintains relative order.
* **Subset**: All possible combinations (power set) of elements.

**🧮 Common Array Techniques**

| **Technique** | **Use Case** |
| --- | --- |
| **Two Pointers** | Pairs, 2 Sum, Sorted array problems |
| **Sliding Window** | Fixed/variable-size windows (max sum, subarray length) |
| **Prefix Sum** | Range queries, subarray sums |
| **Hashing** | Frequency, element existence |
| **Sorting** | Many optimizations (e.g., sort + binary search) |

**🔹 Array Problem Set**

**🔹 Easy Problems**

* Print Alternates
* Linear Search
* Largest/Second Largest Element
* Remove Duplicates from Sorted
* Generate All Subarrays
* Reverse Array
* Rotate Array
* Move Zeroes to End
* Leaders in Array

**🔹 Medium Problems**

* Make arr[i] = i
* Reorder Based on Index
* Sort 0s, 1s, and 2s (**Dutch National Flag** – O(n))
* Minimum Swaps to Sort
* Maximum Subarray Sum (**Kadane’s Algorithm**)
* Product Except Self
* Merge Overlapping Intervals
* Majority Element (**Boyer–Moore Algorithm**)
* Inversion Count (Merge Sort-based)
* Subarray with Given Sum (Sliding window or hashmap)
* 2 Sum / 3 Sum / 4 Sum
* Peak Element (Binary Search)
* Smallest Subarray with Sum > X

**🔹 Hard Problems**

* Trapping Rain Water
* K-th Missing Positive Number
* Stock Buy and Sell – Max K Transactions
* Median in a Stream
* 3 Sum & 4 Sum (All variants)
* Surpasser Count
* Top K Frequent Elements
* Minimum Jumps to Reach End
* Max Value of Sum(i\*arr[i]) after Rotations
* Range Queries using Sparse Table or Mo’s Algorithm

**🧠 Common Formulas & Tricks**

| **Concept** | **Formula / Tip** |
| --- | --- |
| **Subarrays of array size n** | n(n+1)/2 |
| **Sum of all subarrays** | Use prefix sum |
| **Kadane’s Algo** | Maintain max\_ending\_here & max\_so\_far |
| **Trapping Rain Water** | Precompute left\_max, right\_max |
| **Inversion Count** | Count pairs (i, j) such that i < j and arr[i] > arr[j] |
| **Majority Element** | Occurs more than n/2 times |
| **Prefix Sum** | prefix[i] = prefix[i-1] + arr[i] |
| **Equilibrium Index** | Left sum = Right sum |

**📚 Advanced Concepts for CP & MAANG**

| **Topic** | **Notes** |
| --- | --- |
| **MO’s Algorithm** | Offline range query optimization |
| **Sqrt Decomposition** | Dividing array into √n blocks |
| **Sparse Table** | For immutable arrays (RMQ in O(1)) |
| **Merge Sort Tree** | Query order statistics in ranges |
| **Space Optimization using Bits** | Efficient use of memory |

**📝 Summary Table**

| **Topic** | **Status for MAANG** |
| --- | --- |
| Array Basics | ✅ Must Know |
| Easy-Medium Problems | ✅ Practice till speed |
| Sliding Window, Prefix Sum | ✅ Mastery Required |
| 2 Sum, 3 Sum, 4 Sum Variants | ✅ Expected in interviews |
| Kadane’s, Dutch Flag | ✅ Standard Patterns |
| Mo’s, Sparse Table | 🔁 Optional for CP rounds |

**🔗 Quick Practice Resources**

* GfG Top Array Problems
* LeetCode Array Tag
* InterviewBit Arrays

Would you like me to:

* Combine this with the previous notes and convert to a PDF?
* Create a **Notion-ready doc**?
* Make a **cheat sheet version (1-page review)?**

Let me know your format preference next, and I’ll deliver accordingly!

Absolutely, Manoj! Here's a **1-page Cheat Sheet** for **Arrays in DSA (MAANG-focused)** — compact, quick to revise, and organized for interview prep.

**🧠 DSA Cheat Sheet – Arrays (MAANG Level)**

**🔹 Core Properties**

* **Type**: Linear Data Structure
* **Access**: O(1)
* **Search (unsorted)**: O(n)
* **Insertion/Deletion (middle)**: O(n)
* **Use cases**: Base for Stack, Queue, Matrix, HashTable

**🔹 Advantages**

* ✅ Fast random access arr[i]
* ✅ Cache friendly (contiguous memory)
* ✅ Simple implementation

**🔸 Disadvantages**

* ❌ Fixed size
* ❌ Costly insertion/deletion (middle)
* ❌ Searching = O(n) if unsorted

**🛠️ Must-Know Techniques**

| **Technique** | **Used For** |
| --- | --- |
| **Two Pointers** | 2 Sum, Remove duplicates, Pair problems |
| **Sliding Window** | Max sum/k-sized window, Subarray length |
| **Prefix Sum** | Range queries, Subarray sums |
| **Hashing** | Frequencies, Duplicates, Sum problems |
| **Sorting** | Binary Search, Optimization, Pair logic |

**📘 Formulas & Patterns**

| **Problem Type** | **Formula / Pattern** |
| --- | --- |
| Subarrays Count | n(n+1)/2 |
| Max Subarray Sum | **Kadane’s Algo** |
| Trapping Rain Water | min(L\_max, R\_max) - height[i] |
| Majority Element | **Boyer–Moore Voting Algo** |
| Prefix Sum | prefix[i] = prefix[i-1] + arr[i] |
| Inversion Count | Merge Sort based |
| Equilibrium Index | left\_sum == total - left\_sum - arr[i] |

**🔍 Top Interview Problems (Grouped)**

**✅ Easy**

* Reverse Array
* Leaders in Array
* Remove Duplicates (Sorted)
* Rotate Array
* Move Zeroes to End

**🟡 Medium**

* Kadane’s Algo (Max Subarray Sum)
* Sort 0s,1s,2s (Dutch Flag)
* Merge Overlapping Intervals
* Product Except Self
* Subarray with Given Sum
* 2 Sum, 3 Sum, Peak Element

**🔴 Hard**

* Trapping Rain Water
* Minimum Jumps to End
* Kth Missing Positive
* 4 Sum Variants
* Max Sum with Rotations
* Median in Stream
* Mo’s Algorithm / Sqrt Decomposition

**🧮 Complexity Snapshot**

| **Operation** | **Time** |
| --- | --- |
| Access arr[i] | O(1) |
| Search (Unsorted) | O(n) |
| Search (Sorted) | O(log n) |
| Insert/Delete Mid | O(n) |
| Traversal | O(n) |

**⚡ Quick Reminders**

* 🔁 Subarray = contiguous
* 🔀 Subsequence = not necessarily contiguous
* 🔣 Subset = power set (2ⁿ)
* 🧠 Learn patterns, not problems — 2 Sum → 3 Sum → 4 Sum builds up

Would you like this:

* Exported as a **PDF file**?
* Designed into a **stylish printable poster version**?
* Added into your ongoing **Notion doc**?

Let me know and I’ll prepare it right away.

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... M DSA Course DSA Practice Searching Algorithms MCQs on Searching Algorithms Tutorial on Searching Algorithms Linear Search Binary Search Ternary Search Jump Search Sentinel Linear Search Interpolation Search Exponential Search Fibonacci Search Ubiquitous Binary Search Linear Search Vs Binary Search Interpolation Search Vs Binary Search Binary Search Vs Ternary Search Sentinel Linear Search Vs Linear Search Explore GfG Courses Share Your Experiences DSA Tutorial - Learn Data Structures and Algorithms Basics & Prerequisites Data Structures Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Interview Preparation Practice Problem DSA to DevelopmentCourse Searching Algorithms Last Updated : 15 May, 2025 Searching algorithms are essential tools in computer science used to locate specific items within a collection of data. In this tutorial, we are mainly going to focus upon searching in an array. When we search an item in an array, there are two most common algorithms used based on the type of input array. Linear Search : It is used for an unsorted array. It mainly does one by one comparison of the item to be search with array elements. It takes linear or O(n) Time. Binary Search : It is used for a sorted array. It mainly compares the array's middle element first and if the middle element is same as input, then it returns. Otherwise it searches in either left half or right half based on comparison result (Whether the mid element is smaller or greater). This algorithm is faster than linear search and takes O(Log n) time. One more common search technique is Two Pointers Technique where we begin searching from both ends of the array. Library Implementations of Binary Search binary\_search, lower\_bound and upper\_bound in C++ Arrays.binarySearch() in Java Arrays.binarySearch() in Java for Search in subarray Collections.binarySearch() in Java Bisect in Python List.BinarySearch in C# Easy Problems Largest in an Array Second Largest in an array Largest three in an array Missing Number First Repeating Missing and Repeating Count 1’s in a sorted binary array Closest to 0 Sum Pair Pair with the given difference k largest(or smallest) Elements Kth smallest in row and column-wise sorted Common elements in 3 sorted Ceiling in a sorted Floor in a Sorted Maximum in a Bitonic Elements that appear more than n/k times Medium Problems Triplets with zero sum Partition Point Largest pair sum K’th Smallest in Unsorted Array Search an in a sorted and rotated Min in a sorted and rotated Max in a sorted and rotated Peak element Max and min using minimum comparisons Find a Fixed Point in a given array K most frequent words from a file K closest elements 2 Sum – Pair Sum Closest to Target in Sorted Array Closest pair from two sorted arrays Three closest from three sorted arrays Binary Search for Rationals Missing Element in AP Hard Problems Median of two sorted arrays Median of two sorted of different sizes Search in an almost sorted array Search in a sorted infinite array Pair sum in a sorted and rotated array K’th Smallest/Largest Element in Unsorted Array K’th largest element in a stream Best First Search (Informed Search) More Searching Algorithms Sentinel Linear Search Meta Binary Search | One-Sided Binary Search Ternary Search Jump Search Interpolation Search Exponential Search Fibonacci Search The Ubiquitous Binary Search Comparisons Between Different Searching Algorithms Linear Search vs Binary Search Interpolation search vs Binary search Why is Binary Search preferred over Ternary Search? Is Sentinel Linear Search better than normal Linear Search? Quick Links: ‘Practice Problems’ on Searching Top Searching Interview Questions ‘Quizzes’ on Searching Learn Data Structure and Algorithms | DSA Tutorial Comment More info Campus Training Program Next Article Sorting Algorithms course-img 1418k+ interested Geeks Data Structures and Algorithms - Self Paced [Online Course] Explore course-img 112k+ interested Geeks Data Structures & Algorithms in JavaScript - Self Paced Course Explore course-img 344k+ interested Geeks Data Structures & Algorithms in Python - Self Paced Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox -> Similarly this one

**ChatGPT said:**

Awesome, Manoj! Here's a **clean and organized set of notes** for the **Searching Algorithms** section, optimized for **MAANG interview prep** and structured just like our earlier notes.

**📘 DSA Cheat Sheet – Searching Algorithms (MAANG-Level)**

**🔍 What is Searching?**

Searching algorithms are used to find a **target element** within a collection (typically an array). Efficiency depends on whether the array is **sorted or unsorted**.

**🔸 Core Searching Algorithms**

| **Algorithm** | **Use Case** | **Time Complexity** | **Array Type** |
| --- | --- | --- | --- |
| **Linear Search** | Unsorted arrays | O(n) | Unsorted |
| **Binary Search** | Sorted arrays | O(log n) | Sorted |
| **Ternary Search** | Sorted arrays (math/peak) | O(log₃ n) ~ O(log n) | Sorted |
| **Jump Search** | Optimized for sorted | O(√n) | Sorted |
| **Interpolation Search** | Uniformly distributed | O(log log n) avg | Sorted |
| **Exponential Search** | Unbounded / Infinite arrays | O(log i) | Sorted |
| **Fibonacci Search** | Optimized binary variation | O(log n) | Sorted |
| **Sentinel Linear Search** | Faster Linear | O(n) | Unsorted |
| **Ubiquitous Binary Search** | Binary Search Framework | O(log n) | Sorted |

**🧠 Important Binary Search Applications**

* **Floor/Ceiling in Sorted Array**
* **Search in Rotated Sorted Array**
* **Search in Almost Sorted Array**
* **Find Peak Element**
* **Missing Element in AP**
* **Fixed Point (arr[i] = i)**
* **K Closest Elements**
* **Binary Search on Answer (e.g., Min in Rotated)**

**📘 Common Library Methods**

| **Language** | **Method(s)** |
| --- | --- |
| C++ | binary\_search(), lower\_bound(), upper\_bound() |
| Java | Arrays.binarySearch(), Collections.binarySearch() |
| Python | bisect.bisect\_left(), bisect.bisect\_right() |
| C# | List.BinarySearch() |

**✅ Easy Searching Problems**

* Largest / Second Largest / Three Largest
* Missing Number
* First Repeating
* Count 1’s in a Sorted Binary Array
* Kth Smallest in Sorted Matrix
* Ceiling and Floor in Sorted Array
* Pair with Given Difference
* K Largest Elements
* Common Elements in 3 Sorted Arrays

**🟡 Medium Searching Problems**

* Triplets with 0 Sum
* K Closest Elements
* Search in Rotated Sorted Array
* Closest Pair from 2 Sorted Arrays
* K Most Frequent Words
* Max & Min with Min Comparisons
* Partition Point (Lower Bound logic)
* Binary Search for Rationals
* Missing Element in Arithmetic Progression

**🔴 Hard Searching Problems**

| **Problem** | **Technique** |
| --- | --- |
| **Median of Two Sorted Arrays** | Binary Search on Partition |
| **Search in Infinite Array** | Exponential Search + Binary |
| **Search in Almost Sorted Array** | Modified Binary Search |
| **Kth Smallest Element in Stream** | Min-Heap |
| **Pair Sum in Rotated Sorted Array** | Two Pointers (Modulo logic) |
| **Best First Search (Heuristic)** | Graph-based (Informed Search) |

**🔢 Binary Search Variants**

| **Variant** | **Concept** |
| --- | --- |
| **Meta Binary Search** | One-sided binary search for special cases |
| **Ubiquitous Binary Search** | Binary search on monotonic function (e.g., min x such that condition holds) |
| **Binary Search on Answer** | Find min/max feasible value (e.g., min speed, max weight capacity) |

**🔁 Comparison Table**

| **Type** | **Best For** | **Time** | **Sorted Required?** |
| --- | --- | --- | --- |
| Linear | Small unsorted | O(n) | ❌ |
| Binary | Large sorted | O(log n) | ✅ |
| Interpolation | Uniformly spaced | O(log log n) | ✅ |
| Jump | Medium-sized | O(√n) | ✅ |
| Sentinel | Avoid out-of-bound | O(n) | ❌ |

**💡 Search Pattern Tips**

| **Pattern** | **Use Case** |
| --- | --- |
| **Lower Bound** | First element ≥ target |
| **Upper Bound** | First element > target |
| **Find Min/Max** | Use Binary Search on Answer |
| **Closest Element** | Check abs(mid - target) |
| **Rotated Array Search** | Modified Binary + condition check |
| **Infinite Array** | Exponential Index Search |

**📌 Key MAANG Insight**

* **Binary Search mastery is non-negotiable.**
* Most **hard DSA problems reduce to binary search on condition or answer**.
* In interviews, you're expected to:
  + **Write binary search from scratch**
  + **Modify binary search conditions**
  + **Convert brute force to BS-based optimal solution**

Would you like to:

* Merge this with the previous **Array, Algorithm Analysis, and Logic Building** notes into one study PDF?
* Convert all this into **Notion format**?
* Build a **personalized roadmap PDF** from these concepts?

Let me know what format you prefer next!

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... M DSA Course DSA Practice Sorting MCQs on Sorting Tutorial on Sorting Bubble Sort Quick Sort Merge Sort Insertion Sort Selection Sort Heap Sort Sorting Complexities Radix Sort ShellSort Counting Sort Bucket Sort TimSort Bitonic Sort Uses of Sorting Algorithm Explore GfG Courses Share Your Experiences DSA Tutorial - Learn Data Structures and Algorithms Basics & Prerequisites Data Structures Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Interview Preparation Practice Problem DSA to DevelopmentCourse Sorting Algorithms Last Updated : 12 Jun, 2025 A Sorting Algorithm is used to rearrange a given array or list of elements in an order. For example, a given array [10, 20, 5, 2] becomes [2, 5, 10, 20] after sorting in increasing order and becomes [20, 10, 5, 2] after sorting in decreasing order. There exist different sorting algorithms for different different types of inputs, for example a binary array, a character array, an array with a large range of values or an array with many duplicates or a small vs large array. The algorithms may also differ according to output requirements. For example, stable sorting (or maintains original order of equal elements) or not stable. Sorting is provided in library implementation of most of the programming languages. These sorting functions typically are general purpose functions with flexibility of providing the expected sorting order (increasing or decreasing or by a specific key in case of objects). Basics of Sorting Algorithms: Introduction to Sorting Applications of Sorting Sorting Algorithms: Comparison Based : Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort, Cycle Sort, 3-way Merge Sort Non Comparison Based : Counting Sort, Radix Sort, Bucket Sort, TimSort, Comb Sort, Pigeonhole Sort Hybrid Sorting Algorithms : IntroSort, Tim Sort Library Implementations: qsort() in C sort() in C++ STL Arrays.sort() in Java with examples Collections.sort() in Java with Examples Sort a List in Python Sorting in JavaScript Easy Problems on Sorting: Check if an array is Sorted Sort an array of two types Sort a String Sort Each Row of a Matrix Sort a Matrix Sort a Linked List Sort in Wave Form Sort by Frequency Sort from Different Machines Check if any two intervals overlap Missing elements of a range Sort by set bits counts Sort even and odd placed in different orders Sorting Big Integers Sort strings by lengths Merge Two Sorted Arrays Sort when two halves are sorted 2 Sum - Pair in a Sorted Array Intersection of two sorted arrays Union of two sorted arrays Meeting Rooms Medium Problems on Sorting: Minimum Increments to Make Unique Merge Overlapping Intervals Minimum Platforms Closest Pair of Elements Closest Pair of Points Chocolate Distribution Problem Min and Max Amount to Buy All Three Way Partitioning Sort an array of 0s, 1s and 2s Sort a linked list of 0s, 1s and 2s Inversion count K-th Smallest Element K Smallest Elements 3 Sum - Find Any 3 Sum - Closest Triplet Smallest Difference Triplet from Three arrays Merge K Sorted Arrays Merge K Sorted Linked Lists Min Unsorted Subarray to make array sorted Sort a nearly sorted array Sort n numbers in range from 0 to n^2 – 1 Sort an array of 1 to n Sort according to order defined by another Maximum intervals overlap Permutation with worst Case of Merge Sort Minimum swaps to make two arrays identical Permute two arrays such that all pair suns are greater than K Bucket Sort To Sort an Array with Negative Numbers Convert an Array to reduced form using Vector of pairs Check if array can be sorted with conditional swapping of adjacent 4 Sum - Find Any [More problems an 4 Sum are in Hard Section] Hard Problems on Sorting: Merge Without Extra Space Top K Frequent Elements 3 Sum - Distinct Triplets 4 Sum - Distinct Quadruples 4 Sum - All Quadruples 4 Sum - Closest Quadruple Surpasser Counts in an Array Count distinct occurrences as a subsequence Minimum consecutive number subsets Minimum swaps for Binary Tree to BST K-th smallest element after removing some integers from natural numbers Max frequency diff such greater freq item is also is also greater Min swaps to reach permuted array with at most 2 positions left swaps allowed Making Array Elements Same Sort an array after applying an equation Array of strings in sorted order without copying strings Quick Links : ‘Practice Problems’ on Sorting Top Sorting Interview Questions ‘Quizzes’ on Sorting Recommended: Learn Data Structure and Algorithms | DSA Tutorial Comment More info Campus Training Program Next Article Introduction to Recursion course-img 1418k+ interested Geeks Data Structures and Algorithms - Self Paced [Online Course] Explore course-img 112k+ interested Geeks Data Structures & Algorithms in JavaScript - Self Paced Course Explore course-img 344k+ interested Geeks Data Structures & Algorithms in Python - Self Paced Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox -> Similarly this one

**ChatGPT said:**

Absolutely, Manoj! Here's your **MAANG-optimized Sorting Algorithms cheat sheet** based on the GeeksforGeeks content you've shared. This note is crisp, categorized, and interview-focused for fast revision and problem-solving.

**📘 DSA Cheat Sheet – Sorting Algorithms (MAANG-Level)**

**🔍 What is Sorting?**

Sorting is the process of **arranging elements** in a particular order (ascending/descending). Used in:

* Searching optimization
* Duplicates removal
* Scheduling problems
* Range queries
* Data analysis

**🔸 Types of Sorting Algorithms**

**🔹 Comparison-Based Sorting (Compare elements)**

| **Algorithm** | **Time Complexity (Avg)** | **Space** | **Stable** |
| --- | --- | --- | --- |
| **Bubble Sort** | O(n²) | O(1) | ✅ |
| **Selection Sort** | O(n²) | O(1) | ❌ |
| **Insertion Sort** | O(n²) (Best: O(n)) | O(1) | ✅ |
| **Merge Sort** | O(n log n) | O(n) | ✅ |
| **Quick Sort** | O(n log n) (Worst: O(n²)) | O(log n) | ❌ |
| **Heap Sort** | O(n log n) | O(1) | ❌ |
| **Cycle Sort** | O(n²) | O(1) | ❌ |
| **3-Way Merge Sort** | O(n log n) | O(n) | ✅ |

**🔹 Non-Comparison-Based Sorting (No element-to-element comparison)**

| **Algorithm** | **Time Complexity** | **Space** | **Stable** |
| --- | --- | --- | --- |
| **Counting Sort** | O(n + k) | O(k) | ✅ |
| **Radix Sort** | O(nk) | O(n + k) | ✅ |
| **Bucket Sort** | O(n + k) | O(n + k) | ✅ |
| **TimSort** | O(n log n) (Hybrid) | O(n) | ✅ |

**🔹 Hybrid Sorting**

* **TimSort**: Used in Python/Java. Hybrid of Insertion + Merge.
* **IntroSort**: Hybrid of Quick + Heap + Insertion.

**✅ Sorting Algorithm Applications**

* Merge Intervals → **Merge Sort**
* Kth Smallest → **QuickSelect / Heap**
* Top K Frequent Elements → **Heap + Map**
* Count Inversions → **Merge Sort**
* External Sorting → **Merge Sort**
* Sort 0s, 1s, 2s → **Dutch National Flag**
* Stable Sort required → **Merge / Insertion / Counting**

**🧠 Important Interview Patterns**

| **Problem Type** | **Algorithm Hint** |
| --- | --- |
| Sort nearly sorted array | Min-Heap (size k) |
| Count inversions | Merge Sort |
| Min swaps to sort | Cycle Detection (Graphs) |
| Kth smallest/largest | QuickSelect / Heap |
| Sort Linked List | Merge Sort (O(n log n)) |
| Sort elements by frequency | HashMap + Heap |
| Sort big integers | String comparison |

**💡 Sorting Properties**

| **Property** | **Description** |
| --- | --- |
| **Stable** | Equal elements retain relative order |
| **In-Place** | No extra memory used |
| **Adaptive** | Performs better if array is nearly sorted |

**🛠️ Sorting Problems Classification**

**✅ Easy**

* Check if array is sorted
* Sort 0s, 1s
* Sort string
* Merge two sorted arrays
* Sort matrix rows
* Sort by frequency
* Sort by set bit count

**🟡 Medium**

* Sort 0s, 1s, 2s (**Dutch Flag**)
* Inversion Count
* Minimum Platforms
* Merge Overlapping Intervals
* Chocolate Distribution Problem
* Kth Smallest
* Three Way Partition
* Min unsorted subarray to sort

**🔴 Hard**

* **Merge Without Extra Space**
* **Top K Frequent Elements**
* **Kth Smallest Element in Stream**
* **4 Sum Variants**
* **Sort based on equation or key**
* **Sort with only adjacent swaps**

**📘 Sorting Library Functions**

| **Language** | **Function** |
| --- | --- |
| C | qsort() |
| C++ | std::sort(), stable\_sort() |
| Java | Arrays.sort(), Collections.sort() |
| Python | sorted(), .sort(), key= |
| JS | arr.sort((a,b)=>a-b) |

**📌 Time-Space Summary**

| **Algorithm** | **Time (Avg)** | **Space** | **Stable** |
| --- | --- | --- | --- |
| Bubble Sort | O(n²) | O(1) | ✅ |
| Merge Sort | O(n log n) | O(n) | ✅ |
| Quick Sort | O(n log n) | O(log n) | ❌ |
| Heap Sort | O(n log n) | O(1) | ❌ |
| Counting Sort | O(n + k) | O(k) | ✅ |
| Radix Sort | O(nk) | O(n + k) | ✅ |

**🎯 MAANG Interview Tip**

Most **medium-to-hard DSA problems** use sorting as a pre-step. Master:

* **QuickSort partitioning**
* **Merge Sort merging**
* **Heap (Min/Max) usage**
* **Radix/Counting for integer-based sorting**

Would you like:

* All cheat sheets (Logic, Arrays, Searching, Sorting, Algorithm Analysis) combined into a **printable PDF or Notion doc**?
* Create a **study timetable** to revise each topic + solve top problems daily?
* Build flashcards for time complexity and properties of all sorting types?

Let me know! I’ll tailor the next output for your preferred format.

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... 17 M DSA Course DSA Interview Problems on Hash Practice Hash MCQs on Hash Hashing Tutorial Hash Function Index Mapping Collision Resolution Open Addressing Separate Chaining Quadratic probing Double Hashing Load Factor and Rehashing Advantage & Disadvantage ▲ Explore GfG Courses Share Your Experiences DSA Tutorial - Learn Data Structures and Algorithms Basics & Prerequisites Data Structures Getting Started with Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Advanced Interview Preparation Practice Problem DSA to DevelopmentCourse Hashing in Data Structure Last Updated : 09 Jun, 2025 Hashing is a technique used in data structures that efficiently stores and retrieves data in a way that allows for quick access. Hashing involves mapping data to a specific index in a hash table (an array of items) using a hash function. It enables fast retrieval of information based on its key. The great thing about hashing is, we can achieve all three operations (search, insert and delete) in O(1) time on average. Hashing is mainly used to implement a set of distinct items (only keys) and dictionaries (key value pairs). Introduction-to-Hashing Hashing in Data Structure Basics Introduction Applications Separate Chaining for Collision Handling Open Addressing for Collision Handling Easy Problems Check tor Subset Check for Disjoint Check for Equal Fizz Buzz Max distance between two occurrences Duplicate within K Distance Intersection of Two Arrays Union of Two Arrays Most Frequent Element Maximum points on the same line 2 Sum - Find if there is any pair 2 Sum - Count Pairs Count Pairs with Given Diff Only repetitive element from 1 to n-1 Missing elements of a given range Missing Elements from Min to Max of Array Minimum Subsets with distinct elements Minimum Removals for No Common Pairs with given sum from different rows Medium Problems Check If Array Pair Sums Divisible by k Longest subarray with sum divisible by k 3 Sum - Count all triplets with given sum 3 Sum – Find All Triplets with Zero Sum Itinerary from a given list of tickets Longest Subarray having Majority Elements Greater Than K Number of Employees Under every Employee Largest subarray with 0 sum Subarray with given sum Longest Consecutive subsequence Largest Fibonacci Subset Consecutive Subset Partitioning Distincts in every window of size k Insert, delete, search and getRandom Min insertions for a palindrome permutation Maximum possible difference of two subsets of an array Sorting using trivial hash function Smallest subarray with k distinct numbers All pairs (a, b) in an array such that a % b = k Group words with same set of characters k-th distinct (or non-repeating) element in an array. Hard Problems Represent Fraction as String 4 Sum – Count quadruplets 4 Sum – Find all Quadruplets 4 Sum - From four sorted arrays Clone a Binary Tree with Random Pointers Largest subarray with equal number of 0s and 1s Longest Common Sum Span Palindrome Substring Queries Range Queries for Frequencies of array elements Cuckoo Hashing – Worst case O(1) Lookup! Count subarrays having total distinct elements same as original array Maximum array from two given arrays keeping order same Find Sum of all unique sub-array sum for a given array. Recaman’s sequence Length of longest strict bitonic subsequence Find All Duplicate Subtrees Find if there is a rectangle in binary matrix with corners as 1 Quick Links : ‘Practice Problems’ on Hashing Top Hashing Interview Questions ‘Quizzes’ on Hashing Recommended: Learn Data Structure and Algorithms | DSA Tutorial Comment More info Campus Training Program Next Article Linked List Data Structure course-img 344k+ interested Geeks Data Structures & Algorithms in Python - Self Paced Explore course-img 200k+ interested Geeks C Programming Course Online - Learn C with Data Structures Explore course-img 112k+ interested Geeks Data Structures & Algorithms in JavaScript - Self Paced Course Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Thanks for sharing, Manoj! Here's a concise, **interview-focused cheat sheet for Hashing in DSA**, based on your GeeksforGeeks content — ideal for MAANG-level prep and quick revision.

**📘 DSA Cheat Sheet – Hashing (MAANG Level)**

**🔍 What is Hashing?**

Hashing is a technique to map data (keys) to fixed-size values using a **hash function**. It provides:

* 🟢 **Insert/Search/Delete in O(1) on average**
* 🚀 Faster than arrays/lists for lookups
* ⚙️ Backbone for sets, dictionaries/maps, caches, etc.

**🧠 Core Concepts**

| **Concept** | **Description** |
| --- | --- |
| **Hash Function** | Maps a key to an index (e.g., hash(key) % size) |
| **Hash Table** | Array that stores data using hash indices |
| **Load Factor** | n / table\_size; determines when to rehash |
| **Collision** | Two keys map to same index |
| **Rehashing** | Resize & reinsert elements when table is too full |

**🛠️ Collision Resolution Techniques**

**🔸 Separate Chaining**

* Use a linked list at each index
* Easy to implement
* May use extra memory

**🔸 Open Addressing**

* All elements in the same array
* Types:
  + **Linear Probing**: next = (hash + 1) % size
  + **Quadratic Probing**: (hash + i²) % size
  + **Double Hashing**: Use a second hash

| **Method** | **Pros** | **Cons** |
| --- | --- | --- |
| **Chaining** | Handles load well | Extra memory per chain |
| **Open Addressing** | No extra memory | Clustering issues possible |

**🧮 Applications of Hashing**

* Dictionaries, Maps
* Caching (LRU Cache)
* Counting frequencies
* Duplicates detection
* Anagram grouping
* Subarray sum problems
* Pattern matching (Rabin-Karp)

**✅ Easy Hashing Problems**

* ✅ Check for Subset
* ✅ Intersection & Union of Arrays
* ✅ Duplicate within K distance
* ✅ 2 Sum – Find if Pair Exists
* ✅ Max Distance Between Same Elements
* ✅ Missing Elements of a Range
* ✅ Most Frequent Element

**🟡 Medium Hashing Problems**

* 🔹 Longest Subarray with 0 Sum
* 🔹 Subarray with Given Sum
* 🔹 Itinerary Reconstruction
* 🔹 Distinct Elements in Every Window
* 🔹 Insert/Delete/Search/GetRandom in O(1)
* 🔹 Check Array Pair Sums Divisible by K
* 🔹 3 Sum Variants (All Triplets)
* 🔹 Longest Consecutive Subsequence

**🔴 Hard Hashing Problems**

* 🔸 Clone Binary Tree with Random Pointers
* 🔸 Represent Fraction as String
* 🔸 4 Sum Variants (All Quadruplets)
* 🔸 Count Subarrays with Total Distinct = Original Array
* 🔸 Longest Common Span with Same Sum
* 🔸 Range Queries for Element Frequencies
* 🔸 Palindrome Substring Queries
* 🔸 Maximum Array from Two Arrays (Preserving Order)

**🔢 Time Complexities (Average Case)**

| **Operation** | **Time Complexity** |
| --- | --- |
| Insert/Search/Delete | O(1) |
| Worst Case (Collision-heavy) | O(n) |
| Rehashing | O(n) (Amortized) |

**⚡ Library Implementations**

| **Language** | **Hash Table Equivalent** |
| --- | --- |
| C++ | unordered\_map, unordered\_set |
| Java | HashMap, HashSet |
| Python | dict, set |
| JavaScript | Map, Set |

**📌 Key Points for Interviews**

| **Concept** | **Must Know?** |
| --- | --- |
| HashMap Basics | ✅ |
| Collision Handling Techniques | ✅ |
| Open Addressing vs Chaining | ✅ |
| Longest Subarray Sum 0 / K | ✅ |
| Hashing-based Frequency Counting | ✅ |
| Insert/Delete/Search in O(1) | ✅ |
| Hashing in DP/Memoization | ✅ |

**🔍 Practice Keywords (LeetCode / GfG / InterviewBit)**

* Longest Subarray with Sum 0
* Longest Consecutive Sequence
* Group Anagrams
* Count Distinct Elements in Window
* Subarray Sum Equals K
* Design RandomizedSet
* Itinerary Reconstruction

Would you like:

* To **merge this with your full DSA cheat sheet** (Arrays, Searching, Sorting, etc.) into a printable PDF?
* A **flashcard deck** (Anki/Notion-style) for Hashing formulas and problems?
* A **30-day revision plan** with topic-wise high-yield problems?

Let me know your preferred next step, and I’ll build it for you!

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... DSA Course DSA Interview Questions on Array Practice Array MCQs on Array Tutorial on Array Types of Arrays Array Operations Subarrays, Subsequences, Subsets Reverse Array Static Vs Arrays Array Vs Linked List Array | Range Queries Advantages & Disadvantages Explore GfG Courses Share Your Experiences Two Pointers Technique Two Pointers Technique in Different languages Java Program for Two Pointers Technique C Program for Two Pointers Technique Problems based on Two-Pointer Technique Find four elements that sum to a given value | Two-Pointer approach Rearrange an array in maximum minimum form using Two Pointer Technique Pair with largest sum which is less than K in the array Longest increasing sequence by the boundary elements of an Array Two Sum - Pair with given Sum Count of equal value pairs from given two Arrays such that a[i] equals b[j] Move all negative numbers to beginning and positive to end with constant extra space DSA to DevelopmentCourse Two Pointers Technique Last Updated : 16 May, 2025 Two pointers is really an easy and effective technique that is typically used for Two Sum in Sorted Arrays, Closest Two Sum, Three Sum, Four Sum, Trapping Rain Water and many other popular interview questions. Given a sorted array arr (sorted in ascending order) and a target, find if there exists any pair of elements (arr[i], arr[j]) such that their sum is equal to the target. Illustration : Input: arr[] = {10, 20, 35, 50}, target =70 Output: Yes Explanation : There is a pair (20, 50) with given target. Input: arr[] = {10, 20, 30}, target =70 Output : No Explanation : There is no pair with sum 70 Input: arr[] = {-8, 1, 4, 6, 10, 45], target = 16 Output: Yes Explanation : There is a pair (6, 10) with given target. Try it on GfG Practice redirect icon Table of Content Naive Method - O(n^2) Time and O(1) Space Two-Pointer Technique - O(n) time and O(1) space How does this work? More problems based on two pointer technique. Naive Method - O(n^2) Time and O(1) Space The very basic approach is to generate all the possible pairs and check if any of them add up to the target value. To generate all pairs, we simply run two nested loops. C++ #include <bits/stdc++.h> using namespace std; bool twoSum(vector<int> &arr, int target) { int n = arr.size(); // Consider all pairs (arr[i], arr[j]) for (int i = 0; i < n; i++) { for (int j = i + 1; j < n; j++) { // Check if the sum of the current pair // equals the target if (arr[i] + arr[j] == target) { return true; } } } // If no pair is found after checking all return false; } int main() { vector<int> arr = {0, -1, 2, -3, 1}; int target = -2; cout << ((twoSum(arr, target))? "true" : "false"); return 0; } C #include <stdbool.h> #include <stdio.h> // Function to check whether any pair exists // whose sum is equal to the given target value bool twoSum(int arr[], int n, int target){ // Iterate through each element in the array for (int i = 0; i < n; i++){ // For each element arr[i], check every // other element arr[j] that comes after it for (int j = i + 1; j < n; j++){ // Check if the sum of the current pair // equals the target if (arr[i] + arr[j] == target) return true; } } // If no pair is found after checking // all possibilities return false; } int main(){ int arr[] = {0, -1, 2, -3, 1}; int target = -2; int n = sizeof(arr) / sizeof(arr[0]); // Call the twoSum function and print the result if (twoSum(arr, n, target)) printf("true\n"); else printf("false\n"); return 0; } Java class GfG { // Function to check whether any pair exists // whose sum is equal to the given target value static boolean twoSum(int[] arr, int target){ int n = arr.length; // Iterate through each element in the array for (int i = 0; i < n; i++) { // For each element arr[i], check every // other element arr[j] that comes after it for (int j = i + 1; j < n; j++) { // Check if the sum of the current pair // equals the target if (arr[i] + arr[j] == target) { return true; } } } // If no pair is found after checking // all possibilities return false; } public static void main(String[] args){ int[] arr = { 0, -1, 2, -3, 1 }; int target = -2; // Call the twoSum function and print the result if (twoSum(arr, target)) System.out.println("true"); else System.out.println("false"); } } Python # Function to check whether any pair exists # whose sum is equal to the given target value def two\_sum(arr, target): n = len(arr) # Iterate through each element in the array for i in range(n): # For each element arr[i], check every # other element arr[j] that comes after it for j in range(i + 1, n): # Check if the sum of the current pair # equals the target if arr[i] + arr[j] == target: return True # If no pair is found after checking # all possibilities return False arr = [0, -1, 2, -3, 1] target = -2 # Call the two\_sum function and print the result if two\_sum(arr, target): print("true") else: print("false") C# using System; class GfG { // Function to check whether any pair exists // whose sum is equal to the given target value static bool TwoSum(int[] arr, int target) { int n = arr.Length; // Iterate through each element in the array for (int i = 0; i < n; i++) { // For each element arr[i], check every // other element arr[j] that comes after it for (int j = i + 1; j < n; j++) { // Check if the sum of the current pair // equals the target if (arr[i] + arr[j] == target) { return true; } } } // If no pair is found after checking // all possibilities return false; } static void Main() { int[] arr = { 0, -1, 2, -3, 1 }; int target = -2; // Call the TwoSum function and print the result if (TwoSum(arr, target)) Console.WriteLine("true"); else Console.WriteLine("false"); } } JavaScript // Function to check whether any pair exists // whose sum is equal to the given target value function twoSum(arr, target) { let n = arr.length; // Iterate through each element in the array for (let i = 0; i < n; i++) { // For each element arr[i], check every // other element arr[j] that comes after it for (let j = i + 1; j < n; j++) { // Check if the sum of the current pair // equals the target if (arr[i] + arr[j] === target) { return true; } } } // If no pair is found after checking // all possibilities return false; } let arr = [0, -1, 2, -3, 1]; let target = -2; // Call the twoSum function and print the result if (twoSum(arr, target)) console.log("true"); else console.log("false"); Output true Better Approaches - Binary Search and Hashing We can use more methods like Binary Search and Hashing to solve this problem (Please refer Two Sum article for details) in better time complexity but Two Pointer Technique is the best solution for this problem that works well for sorted arrays. Two-Pointer Technique - O(n) time and O(1) space The idea of this technique is to begin with two corners of the given array. We use two index variables left and right to traverse from both corners. Initialize: left = 0, right = n - 1 Run a loop while left < right, do the following inside the loop Compute current sum, sum = arr[left] + arr[right] If the sum equals the target, we’ve found the pair. If the sum is less than the target, move the left pointer to the right to increase the sum. If the sum is greater than the target, move the right pointer to the left to decrease the sum. Illustration: two-pointer-tech-1.webptwo-pointer-tech-1.webp C++ // CPP program demonstrate working of the two // pointer technique #include <bits/stdc++.h> using namespace std; bool twoSum(vector<int> &arr, int target){ int left = 0, right = arr.size() - 1; while (left < right){ int sum = arr[left] + arr[right]; if (sum == target) return true; // Move toward a higher sum else if (sum < target) left++; // Move toward a lower sum else right--; } // If no pair found return false; } int main(){ vector<int> arr = {-3, -1, 0, 1, 2}; int target = -2; if (twoSum(arr, target)) cout << "true"; else cout << "false"; return 0; } C #include <stdbool.h> #include <stdio.h> #include <stdlib.h> // Comparison function for qsort int compare(const void \*a, const void \*b){ return (\*(int \*)a - \*(int \*)b); } // Function to check whether any pair exists // whose sum is equal to the given target value bool twoSum(int arr[], int n, int target){ // Sort the array qsort(arr, n, sizeof(int), compare); int left = 0, right = n - 1; // Iterate while left pointer is less than right while (left < right){ int sum = arr[left] + arr[right]; // Check if the sum matches the target if (sum == target) return true; else if (sum < target) left++; // Move left pointer to the right else right--; // Move right pointer to the left } // If no pair is found return false; } int main(){ int arr[] = {0, -1, 2, -3, 1}; int target = -2; int n = sizeof(arr) / sizeof(arr[0]); // Call the twoSum function and print the result if (twoSum(arr, n, target)) printf("true\n"); else printf("false\n"); return 0; } Java import java.util.Arrays; class GfG { // Function to check whether any pair exists // whose sum is equal to the given target value static boolean twoSum(int[] arr, int target){ // Sort the array Arrays.sort(arr); int left = 0, right = arr.length - 1; // Iterate while left pointer is less than right while (left < right) { int sum = arr[left] + arr[right]; // Check if the sum matches the target if (sum == target) return true; else if (sum < target) left++; // Move left pointer to the right else right--; // Move right pointer to the left } // If no pair is found return false; } public static void main(String[] args){ int[] arr = { 0, -1, 2, -3, 1 }; int target = -2; // Call the twoSum function and print the result if (twoSum(arr, target)) { System.out.println("true"); } else { System.out.println("false"); } } } Python # Function to check whether any pair exists # whose sum is equal to the given target value def two\_sum(arr, target): # Sort the array arr.sort() left, right = 0, len(arr) - 1 # Iterate while left pointer is less than right while left < right: sum = arr[left] + arr[right] # Check if the sum matches the target if sum == target: return True elif sum < target: left += 1 # Move left pointer to the right else: right -= 1 # Move right pointer to the left # If no pair is found return False arr = [0, -1, 2, -3, 1] target = -2 # Call the two\_sum function and print the result if two\_sum(arr, target): print("true") else: print("false") C# using System; using System.Linq; class GfG { // Function to check whether any pair exists // whose sum is equal to the given target value static bool TwoSum(int[] arr, int target){ // Sort the array Array.Sort(arr); int left = 0, right = arr.Length - 1; // Iterate while left pointer is less than right while (left < right) { int sum = arr[left] + arr[right]; // Check if the sum matches the target if (sum == target) return true; else if (sum < target) left++; // Move left pointer to the right else right--; // Move right pointer to the left } // If no pair is found return false; } static void Main(){ int[] arr = { 0, -1, 2, -3, 1 }; int target = -2; // Call the TwoSum function and print the result if (TwoSum(arr, target)) Console.WriteLine("true"); else Console.WriteLine("false"); } } JavaScript // Function to check whether any pair exists // whose sum is equal to the given target value function twoSum(arr, target) { // Sort the array arr.sort((a, b) => a - b); let left = 0, right = arr.length - 1; // Iterate while left pointer is less than right while (left < right) { let sum = arr[left] + arr[right]; // Check if the sum matches the target if (sum === target) return true; else if (sum < target) left++; // Move left pointer to the right else right--; // Move right pointer to the left } // If no pair is found return false; } let arr = [ 0, -1, 2, -3, 1 ]; let target = -2; // Call the twoSum function and print the result if (twoSum(arr, target)) { console.log("true"); } else { console.log("false"); } Output true Time Complexity: O(n) as the loops runs at most n times. We either increase left, or decrease right or stop the loop. Auxiliary Space: O(1) How does this work? We need to prove that we never miss a valid pair. Case 1 ( When we increment left ) In this case we simply ignore current arr[left] and move to the next element by doing left++. We do this when arr[left] + arr[right] is smaller than the target. The reason this step is safe is, if arr[left] is giving a smaller value than sum, then it will given even much less values for the elements before arr[right]. Now how about the elements after arr[right]? Note that we moved right when we were sure that no pair can be formed with the current right (arr[right] was too high), so arr[left] can not form a pair with those values also. Case 2 (When we decrement right) We can use the same reasoning (as we discussed for left) to prove that we never miss out a valid pair. More problems based on two pointer technique. Top Problems for Two Pointers Coding Practice on Two Pointer Algorithms Comment More info Campus Training Program Next Article Java Program for Two Pointers Technique course-img 1418k+ interested Geeks Data Structures and Algorithms - Self Paced [Online Course] Explore course-img 112k+ interested Geeks Data Structures & Algorithms in JavaScript - Self Paced Course Explore course-img 344k+ interested Geeks Data Structures & Algorithms in Python - Self Paced Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Thanks for sharing the full reference, Manoj! Based on the GeeksforGeeks article you provided, here’s a **clean, MAANG-ready cheat sheet for the Two Pointers Technique** — formatted for **quick revision, interview prep**, and structured learning.

**📘 DSA Cheat Sheet – Two Pointers Technique (MAANG Level)**

**🧠 What is Two Pointers Technique?**

A powerful technique for solving problems efficiently by maintaining **two indices (pointers)** that move toward each other or in the same direction depending on the condition.

✅ Most commonly used on **sorted arrays or strings**

**🔁 Common Use Cases**

| **Use Case** | **Description** |
| --- | --- |
| **Two Sum (Sorted Array)** | Find pair with a target sum |
| **Three/Four Sum** | Extend 2-pointers with loop(s) |
| **Trapping Rain Water** | Water stored between bars |
| **Reverse Array / String** | Swap front and back |
| **Segregate Negatives/Positives** | Move all negatives to front |
| **Remove Duplicates** | Keep only unique elements |
| **Longest Subsequence or Window** | With condition like sum or distinct elements |

**⚙️ Working Logic**

**For Sorted Array – Find if two numbers sum to a target:**

int left = 0, right = n - 1; while (left < right) { int sum = arr[left] + arr[right]; if (sum == target) return true; else if (sum < target) left++; else right--; }

**🚀 Naive Approach vs Two Pointers**

| **Approach** | **Time Complexity** | **Space Complexity** |
| --- | --- | --- |
| **Naive (Nested Loops)** | O(n²) | O(1) |
| **Two Pointers** | O(n) | O(1) |

📌 **Condition**: Two Pointers requires **sorted array** for most problems.

**💡 Key Cases That Make It Efficient**

* **Sorted input**
* **Linear scanning needed**
* **Looking for sum, difference, or conditions between two elements**

**🧪 Illustration Example**

arr = [-8, 1, 4, 6, 10, 45] target = 16 left, right = 0, len(arr) - 1 while left < right: current\_sum = arr[left] + arr[right] if current\_sum == target: return True elif current\_sum < target: left += 1 else: right -= 1

✅ Output: True (because 6 + 10 = 16)

**✅ Popular Interview Problems Using Two Pointers**

**🔹 Easy to Medium:**

* Reverse an array/string
* Remove duplicates from sorted array
* Move zeroes to the end
* Two sum (sorted)
* 3 sum
* Pair with sum less than K
* Find intersection of two sorted arrays
* Merge two sorted arrays
* Container with most water

**🔹 Advanced:**

* Trapping Rain Water
* 4 Sum
* Longest substring with at most K distinct characters
* Partitioning arrays based on conditions
* Rearranging max-min (alternating) order

**🧠 Tips to Master Two Pointers**

| **Situation** | **Action** |
| --- | --- |
| **Sum too small** | Increment left pointer |
| **Sum too large** | Decrement right pointer |
| **Valid pair found** | Return or continue for count |
| **Strings** | Useful for palindrome check, window problems |
| **Arrays** | Efficient merging, pairing, subarray scanning |

**📌 Two Pointers Variants**

| **Variant** | **Description** |
| --- | --- |
| **Same direction** | Used when scanning entire array (sliding window) |
| **Opposite direction** | For problems on sorted arrays (like Two Sum) |
| **Slow-Fast Pointers** | Cycle detection (Floyd’s algo), Linked Lists |

**🛠️ Time & Space Complexity**

| **Use Case** | **Time** | **Space** |
| --- | --- | --- |
| Sorted Two Sum | O(n) | O(1) |
| 3 Sum (with sorting) | O(n²) | O(1) |
| 4 Sum (with sorting) | O(n³) | O(1) |
| Remove Duplicates | O(n) | O(1) |
| Trapping Rain Water | O(n) | O(1) |

**🔍 Top Problems to Practice**

* Two Sum (Sorted)
* 3 Sum
* Container With Most Water
* Move Zeroes
* Trapping Rain Water
* Reverse Words in a String

Would you like me to:

* Merge this with your other DSA notes (Array, Hashing, Sorting, Searching)?
* Create a PDF or Notion-style reference guide?
* Build a **30-day Two-Pointers mastery challenge** with daily LeetCode links?

Let me know how you'd like to use this next!

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... 17 M DSA Course DSA Tutorial Data Structures Algorithms Array Strings Linked List Stack Queue Tree Graph Searching Sorting Recursion Dynamic Programming Binary Tree Binary Search Tree Heap Hashing Divide & Conquer Mathematical Geometric Bitwise Greedy Backtracking Branch and Bound Matrix Pattern Searching Randomized Explore GfG Courses Share Your Experiences DSA Tutorial - Learn Data Structures and Algorithms Array Data Structure Guide String in Data Structure Matrix Data Structure Searching Algorithms Sorting Algorithms Hashing in Data Structure Two Pointers Technique Sliding Window Technique Prefix Sum Array - Implementation and Applications Linked List Data Structure Stack Data Structure Queue Data Structure Recursive Algorithms Tree Data Structure Heap Data Structure Greedy Algorithms Dynamic Programming or DP Divide and Conquer Algorithm Graph Algorithms Competitive Programming - A Complete Guide DSA to DevelopmentCourse Sliding Window Technique Last Updated : 20 May, 2025 Sliding Window Technique is a method used to solve problems that involve subarray or substring or window. The main idea is to use the results of previous window to do computations for the next window. This technique is commonly used in algorithms like finding subarrays with a specific sum, finding the longest substring with unique characters, or solving problems that require a fixed-size window to process elements efficiently. Table of Content Example Problem - Maximum Sum of a Subarray with K Elements How to use Sliding Window Technique? How to Identify Sliding Window Problems? More Example Problems Example Problem - Maximum Sum of a Subarray with K Elements Given an array and an integer k, we need to calculate the maximum sum of a subarray having size exactly k. Input : arr[] = {100, 200, 300, 400}, k = 2 Output : 700 We get maximum sum by considering the subarray [300, 400] Input : arr[] = {1, 4, 2, 10, 23, 3, 1, 0, 20}, k = 4 Output : 39 We get maximum sum by adding subarray {4, 2, 10, 23} of size 4. Input : arr[] = {2, 3}, k = 3 Output : Invalid There is no subarray of size 3 as size of whole array is 2. Naive Approach - O(n k) Time and O(1) Space We run two nest loops and compute sums of all subarrays of size k #include <iostream> #include <vector> #include <climits> ​ using namespace std; ​ // Returns maximum sum in a subarray of size k. int maxSum(vector<int>& arr, int k) { int n = arr.size(); int max\_sum = INT\_MIN; ​ // Consider all blocks starting with i for (int i = 0; i <= n - k; i++) { int current\_sum = 0; ​ // Calculate sum of current subarray of size k for (int j = 0; j < k; j++) current\_sum += arr[i + j]; ​ // Update result if required max\_sum = max(current\_sum, max\_sum); } ​ return max\_sum; } ​ // Driver code int main() { vector<int> arr = {1, 4, 2, 10, 2, 3, 1, 0, 20}; int k = 4; ​ cout << maxSum(arr, k) << endl; return 0; } Output 24 Using the Sliding Window Technique - O(n) Time and O(1) Space We compute the sum of the first k elements out of n terms using a linear loop and store the sum in variable window\_sum. Then we will traverse linearly over the array till it reaches the end and simultaneously keep track of the maximum sum. To get the current sum of a block of k elements just subtract the first element from the previous block and add the last element of the current block. The below representation will make it clear how the window slides over the array. Consider an array arr[] = {5, 2, -1, 0, 3} and value of k = 3 and n = 5 This is the initial phase where we have calculated the initial window sum starting from index 0 . At this stage the window sum is 6. Now, we set the maximum\_sum as current\_window i.e 6. Now, we slide our window by a unit index. Therefore, now it discards 5 from the window and adds 0 to the window. Hence, we will get our new window sum by subtracting 5 and then adding 0 to it. So, our window sum now becomes 1. Now, we will compare this window sum with the maximum\_sum. As it is smaller, we won't change the maximum\_sum. Similarly, now once again we slide our window by a unit index and obtain the new window sum to be 2. Again we check if this current window sum is greater than the maximum\_sum till now. Once, again it is smaller so we don't change the maximum\_sum. Therefore, for the above array our maximum\_sum is 6. Below is the code for above approach: #include <iostream> #include <vector> using namespace std; ​ // Returns maximum sum in a subarray of size k. int maxSum(vector<int>& arr, int k) { int n = arr.size(); ​ // n must be greater if (n <= k) { cout << "Invalid"; return -1; } ​ // Compute sum of first window of size k int max\_sum = 0; for (int i = 0; i < k; i++) max\_sum += arr[i]; ​ // Compute sums of remaining windows by // removing first element of previous // window and adding last element of // current window. int window\_sum = max\_sum; for (int i = k; i < n; i++) { window\_sum += arr[i] - arr[i - k]; max\_sum = max(max\_sum, window\_sum); } ​ return max\_sum; } ​ // Driver code int main() { vector<int> arr = { 1, 4, 2, 10, 2, 3, 1, 0, 20 }; int k = 4; cout << maxSum(arr, k); return 0; } Output 24 How to use Sliding Window Technique? There are basically two types of sliding window: 1. Fixed Size Sliding Window: The general steps to solve these questions by following below steps: Find the size of the window required, say K. Compute the result for 1st window, i.e. include the first K elements of the data structure. Then use a loop to slide the window by 1 and keep computing the result window by window. 2. Variable Size Sliding Window: The general steps to solve these questions by following below steps: In this type of sliding window problem, we increase our right pointer one by one till our condition is true. At any step if our condition does not match, we shrink the size of our window by increasing left pointer. Again, when our condition satisfies, we start increasing the right pointer and follow step 1. We follow these steps until we reach to the end of the array. How to Identify Sliding Window Problems? These problems generally require Finding Maximum/Minimum Subarray, Substrings which satisfy some specific condition. The size of the subarray or substring ‘k’ will be given in some of the problems. These problems can easily be solved in O(n2) time complexity using nested loops, using sliding window we can solve these in O(n) Time Complexity. Required Time Complexity: O(n) or O(n log n) Constraints: n <= 106 More Example Problems Top Problems on Sliding Window Technique Practice Questions on Sliding Window Comment More info Campus Training Program Next Article Prefix Sum Array - Implementation and Applications course-img 640k+ interested Geeks DSA to Development: A Complete Guide Explore course-img 29k+ interested Geeks GATE CSE 2026 Online Course [Live Weekday Classes ] Explore course-img 4k+ interested Geeks GATE CSE + DA 2026 Live Course [with Placement Preparation] Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Great! You’ve shared the full content from GFG’s **Sliding Window Technique**, which is essential for solving many MAANG-level problems efficiently.

Here’s a clean, **DSA MAANG-Ready Notes version** of the **Sliding Window Technique** based on what you shared, optimized for quick revision, interviews, and implementation:

**📘 DSA Cheat Sheet – Sliding Window Technique (MAANG Level)**

**🧠 What is Sliding Window Technique?**

A technique used to **reduce time complexity** from O(n²) to O(n) by **reusing the result of previous computations** as we slide a fixed or variable-size window across the array/string.

**📌 Types of Sliding Window**

| **Type** | **Use Case Example** | **Notes** |
| --- | --- | --- |
| **Fixed-size window (K)** | Max sum of subarray of size K | Pre-defined window size |
| **Variable-size window** | Longest substring with unique characters | Window size changes dynamically based on condition |

**🔁 Fixed Size Sliding Window – Steps**

1. Calculate the sum of the **first**k**elements** (initial window)
2. Slide the window by 1 element at a time:
   * Subtract the element that is **leaving**
   * Add the element that is **entering**
3. Track and update the **maximum/minimum** as needed

**✅ Example Problem: Maximum Sum of Subarray of Size K**

**Problem**:  
Given arr[] = {1, 4, 2, 10, 2, 3, 1, 0, 20}, find **maximum sum** of a subarray of size k = 4

**Naive Approach**:

*// Time: O(n\*k)* for (int i = 0; i <= n - k; i++) { int curr\_sum = 0; for (int j = 0; j < k; j++) curr\_sum += arr[i + j]; max\_sum = max(max\_sum, curr\_sum); }

**Optimized Sliding Window Approach**:

*// Time: O(n), Space: O(1)* int maxSum(vector<int>& arr, int k) { int window\_sum = 0; for (int i = 0; i < k; i++) window\_sum += arr[i]; int max\_sum = window\_sum; for (int i = k; i < arr.size(); i++) { window\_sum += arr[i] - arr[i - k]; max\_sum = max(max\_sum, window\_sum); } return max\_sum; }

✅ **Output**: 24

**🧠 How to Identify Sliding Window Problems?**

| **Clues to Look For** |
| --- |
| "Maximum/minimum sum" of subarray or substring |
| "Longest" or "smallest" subarray that satisfies a condition |
| “Exactly K” or “at most/at least K” elements |
| Constraints allow O(n) or O(n log n) only (n ≤ 10⁶) |

**🔄 Variable Size Sliding Window – Pattern**

Used when:

* Window size isn't fixed
* You are given a **condition** (e.g., sum < X, count of distinct elements ≤ K)

**Steps**:

1. Move the right pointer and expand window
2. If condition fails, move left pointer to shrink window
3. Keep updating answer when condition is valid

**📌 Fixed vs Variable Sliding Window Summary**

| **Feature** | **Fixed Window** | **Variable Window** |
| --- | --- | --- |
| Window size known | Yes (K is given) | No (based on condition) |
| Pointer movement | Right moves, Left fixed | Both Left & Right can move |
| Examples | Max Sum of K elements | Longest Substring with K distinct |

**🚀 Top Problems for Practice (MAANG Level)**

| **Problem** | **Type** | **Link** |
| --- | --- | --- |
| Maximum Sum Subarray of Size K | Fixed | LeetCode 643 |
| Longest Substring Without Repeating Characters | Variable | LeetCode 3 |
| Minimum Size Subarray Sum | Variable | LeetCode 209 |
| Longest Repeating Character Replacement | Variable | LeetCode 424 |
| Sliding Window Maximum | Fixed (with Heap/Deque) | LeetCode 239 |

**📚 Key Tips to Master Sliding Window**

* Always try **brute-force O(n²)** first to identify overlapping work
* Visualize the window on paper (helps with variable-sized logic)
* Maintain window state smartly: use maps/sets if needed
* Keep track of max/min/length/condition while updating window

**📈 Complexities**

| **Operation** | **Time** | **Space** |
| --- | --- | --- |
| Max Sum Subarray (Fixed K) | O(n) | O(1) |
| Longest Unique Substring | O(n) | O(K) |
| Max in Sliding Window | O(n) | O(k) with deque |

Would you like me to:

* Merge this into your DSA notes for **Two Pointers** and **Sliding Window**?
* Format it into a **PDF / Notion page / printable sheet**?
* Create a **10-day LeetCode practice schedule** for mastering this technique?

Let me know how you’d like to continue — this will be your strong foundation for MAANG-level DSA.

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... DSA Course DSA Interview Problems on String Practice String MCQs on String Tutorial on String String Operations Sort String Substring & Subsequence Iterate String Reverse String Rotate String String Concatenation Compare Strings KMP Algorithm Boyer-Moore Algorithm Rabin-Karp Algorithm Z Algorithm String Guide for CP Explore GfG Courses Share Your Experiences String in Data Structure Introduction to Strings - Data Structure and Algorithm Tutorials Applications, Advantages and Disadvantages of String Subsequence and Substring Storage for Strings in C Strings in different language Basic operations on String Binary String Substring and Subsequence Palindrome Easy problems on String Intermediate problems on String Hard problems on String Top 50 String Coding Problems for Interviews DSA to DevelopmentCourse Introduction to Strings - Data Structure and Algorithm Tutorials Last Updated : 22 Feb, 2025 Strings are sequences of characters. The differences between a character array and a string are, a string is terminated with a special character ‘\0’ and strings are typically immutable in most of the programming languages like Java, Python and JavaScript. Below are some examples of strings: "geeks" , "for", "geeks", "GeeksforGeeks", "Geeks for Geeks", "123Geeks", "@123 Geeks" How Strings are represented in Memory? In C, a string can be referred to either using a character pointer or as a character array. When strings are declared as character arrays, they are stored like other types of arrays in C. String literals (assigned to pointers) are immutable in C and C++. In C++, strings created using string class are mutable and internally represented as arrays. In Python, Java and JavaScript, strings characters are stored at contiguous locations (like arrays). String How to Declare Strings in various languages? C: Strings are declared as character arrays or pointers and must end with a null character (\0) to indicate termination. C++: Supports both C-style character arrays and the std::string class, which provides built-in functions for string manipulation. Java: Strings are immutable objects of the String class, meaning their values cannot be modified once assigned. Python: Strings are dynamic and can be declared using single, double, or triple quotes, making them flexible for multi-line text handling. JavaScript: Strings are primitive data types and can be defined using single, double, or template literals (backticks), allowing for interpolation. C#: Uses the string keyword, which represents an immutable sequence of characters, similar to Java. There is no character type on Python and JavaScript and a single character is also considered as a string. Below is the representation of strings in various languages: // C++ program to demonstrate String // using Standard String representation ​ #include <iostream> #include <string> using namespace std; ​ int main() { ​ // Declare and initialize the string string str1 = "Welcome to GeeksforGeeks!"; ​ // Initialization by raw string string str2("A Computer Science Portal"); ​ // Print string cout << str1 << endl << str2; ​ return 0; } Are Strings Mutable in Different Languages? In C/C++, string literals (assigned to pointers) are immutable. In C++, string objects are mutable. In Python, Java and JavaScript, strings are immutable. #include <iostream> using namespace std; ​ int main() { const char\* str = "Hello, world!"; str[0] = 'h'; // Error : Assignment to read only cout << str; return 0; } General Operations performed on String Here we are providing you with some must-know concepts of string: Length of String : The length of a string refers to the total number of characters present in it, including letters, digits, spaces, and special characters. It is a fundamental property of strings in any programming language and is often used in various operations such as validation, manipulation, and comparison. Search a Character : Searching for a character in a string means finding the position where a specific character appears. If the character is present multiple times, you might need to find its first occurrence, last occurrence, or all occurrences. Check for Substring : Checking for a substring means determining whether a smaller sequence of characters exists within a larger string. A substring is a continuous part of a string, and checking for its presence is a common operation in text processing, search algorithms, and data validation. Insert a Character : Inserting a character into a string means adding a new character at a specific position while maintaining the original order of other characters. Since strings are immutable in many programming languages, inserting a character usually involves creating a new modified string with the desired character placed at the specified position. Delete a Character : Deleting a character from a string means removing a specific character at a given position while keeping the remaining characters intact. Since strings are immutable in many programming languages, this operation usually involves creating a new string without the specified character. Check for Same Strings : Checking if two strings are the same means comparing them character by character to determine if they are identical in terms of length, order, and content. If every character in one string matches exactly with the corresponding character in another string, they are considered the same. String Concatenation : String concatenation is the process of joining two or more strings together to form a single string. This is useful in text processing, formatting messages, constructing file paths, or dynamically creating content. Reverse a String : Reversing a string means arranging its characters in the opposite order while keeping their original positions intact in the reversed sequence. This operation is commonly used in text manipulation, data encryption, and algorithm challenges. Rotate a String Rotating a string means shifting its characters to the left or right by a specified number of positions while maintaining the order of the remaining characters. The characters that move past the boundary wrap around to the other side. Check for Palindrome : Checking for a palindrome means determining whether a string reads the same forward and backward. A palindrome remains unchanged when reversed, making it a useful concept in text processing, algorithms, and number theory. Introduction to Strings - Data Structure and Algorithm Tutorials - FAQs Is string a linear data structure? Yes, string is a linear data structure. Where are strings used? It is used to store the sequence of characters. Is string a data type? A string is generally considered a data type and is often implemented as an array data structure of bytes (or words) that stores a sequence of elements, typically characters, using some character encoding. Why is text called string? Text are also called string because it consists of sequence of characters like string. What are characters in a string? Each digit in a string is a character and character is a single visual object used to represent text, numbers, or symbols. Search a Character Comment More info Campus Training Program Next Article Applications, Advantages and Disadvantages of String course-img 1417k+ interested Geeks Data Structures and Algorithms - Self Paced [Online Course] Explore course-img 112k+ interested Geeks Data Structures & Algorithms in JavaScript - Self Paced Course Explore course-img 344k+ interested Geeks Data Structures & Algorithms in Python - Self Paced Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Thanks, Manoj! Based on the content you’ve provided from the **GeeksforGeeks Strings Tutorial**, here’s a **cleaned, structured, and MAANG-focused DSA Notes format** for your string concepts. These notes are optimized for **interview prep and quick revision**:

**📘 DSA MAANG Notes – Strings**

**🔤 What is a String?**

A **string** is a sequence of characters. It is typically **immutable** in high-level languages (e.g., Python, Java, JS) and **mutable** when using character arrays in C++.

**Examples**:  
"geeks", "123Geeks", "@123 Geeks", "GeeksforGeeks"

**🧠 Internal Representation**

| **Language** | **Representation** | **Mutability** |
| --- | --- | --- |
| **C** | Character array or char\*, ends with \0 | Immutable (for literals) |
| **C++** | char[] or std::string | std::string is mutable |
| **Java** | String class (internally char[]) | Immutable |
| **Python** | Internally an array of characters | Immutable |
| **JavaScript** | Primitive type (UTF-16 encoded) | Immutable |
| **C#** | string class | Immutable |

**⚙️ Basic String Operations**

| **Operation** | **Description** |
| --- | --- |
| Length | Number of characters in the string |
| Search | Find first/last/all occurrences of a character |
| Check Substring | Find if a smaller sequence exists in the main string |
| Insert Character | Add a character at a specified position |
| Delete Character | Remove a character at a specified index |
| Compare Strings | Lexicographical (dictionary order) comparison |
| Concatenate | Join two or more strings |
| Reverse | Flip the string from end to start |
| Rotate | Circular shift of characters (left/right) |
| Check Palindrome | Same forwards and backwards |

**🧪 Sample Code (C++)**

#include <iostream> #include <string> using namespace std; int main() { string s = "Welcome to GFG!"; cout << "Length: " << s.length() << endl; cout << "First char: " << s[0] << endl; string rev = string(s.rbegin(), s.rend()); cout << "Reverse: " << rev << endl; }

**🏁 Mutable vs Immutable**

| **Language** | **Mutable Strings?** |
| --- | --- |
| C | No (char\* literals) |
| C++ | Yes (std::string) |
| Java | No (String), but StringBuilder is mutable |
| Python | No (use slicing to "modify") |
| JavaScript | No |
| C# | No |

**🧩 Substring vs Subsequence**

| **Feature** | **Substring** | **Subsequence** |
| --- | --- | --- |
| Definition | Continuous part | Can skip characters |
| Example | "abc" from "abcd" | "acd" from "abcd" |

**💻 Advanced String Algorithms (Used in MAANG Interviews)**

| **Algorithm** | **Use Case** |
| --- | --- |
| **KMP** | Pattern searching in O(n + m) |
| **Boyer-Moore** | Pattern matching using bad character heuristic |
| **Rabin-Karp** | Pattern search with hashing |
| **Z Algorithm** | Find all pattern occurrences in linear time |

**🧠 Common Interview Problems on Strings**

**Easy:**

* Reverse a string
* Check for Palindrome
* Remove duplicate characters
* Check anagram

**Medium:**

* Longest substring without repeating characters
* Count and say
* Longest Palindromic Substring
* Group Anagrams

**Hard:**

* Wildcard Matching
* Edit Distance
* Minimum Window Substring
* Word Break

**✅ FAQs**

* **Is String a Linear DS?** → Yes.
* **Is String a data type?** → Yes, a collection of characters (internally arrays).
* **Why is it called a String?** → Like beads in a necklace (sequence).
* **Characters in a String?** → Individual letters, digits, symbols.

**📌 MAANG Tip:**

* Always **dry-run** problems like longest substring, anagrams, and palindromes.
* Focus on using **HashMap**, **Set**, **Two Pointers**, **Sliding Window**, and **DP** in string problems.

Would you like me to:

* Merge this with your **Array/Hashing/Sliding Window notes**?
* Generate a **PDF / Notion format**?
* Give you a **daily 15-string-problems LeetCode/Codeforces challenge plan**?

Let me know how you'd like to proceed next. You're building a powerful base for MAANG-level mastery.

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... 17 M DSA Course DSA Interview Problems on Recursion Practice Recursion MCQs on Recursion Recursion Tutorial Recursive Function Recursion vs Iteration Types of Recursions Tail Recursion Josephus Problem Tower of Hanoi Check Palindrome ▲ Explore GfG Courses Share Your Experiences DSA Tutorial - Learn Data Structures and Algorithms Array Data Structure Guide String in Data Structure Matrix Data Structure Searching Algorithms Sorting Algorithms Hashing in Data Structure Two Pointers Technique Sliding Window Technique Prefix Sum Array - Implementation and Applications Linked List Data Structure Stack Data Structure Queue Data Structure Recursive Algorithms Tree Data Structure Heap Data Structure Greedy Algorithms Dynamic Programming or DP Divide and Conquer Algorithm Graph Algorithms Competitive Programming - A Complete Guide DSA to DevelopmentCourse Recursive Algorithms Last Updated : 29 May, 2025 Recursion is technique used in computer science to solve big problems by breaking them into smaller, similar problems. The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily. Basics Introduction to Recursion Recursion vs Iteration Finite and Infinite Recursion What is Tail Recursion Tail Recursion Optimization Recursion in Different Languages: Recursion in Python Recursion in Java Recursion in C# Recursion in JavaScript Easy Problems Print 1 to n without loop Print n to 1 without loop Mean of Array using Recursion Sum of natural numbers using recursion Decimal to binary number using recursion Sum of array elements using recursion Print reverse of a string using recursion Program for length of a string using recursion Sum of digit of a number using recursion Tail recursion to calculate sum of array elements. Program to print first n Fibonacci Numbers | Set 1 Factorial of a number Array Min and Max using Recursion Palindrome Check using Recursion Count Set-bits using Recursion Fibonacci Series in reverse order using Recursion Medium Problems Recursively remove all adjacent duplicates Sort the Queue using Recursion Reversing a queue using recursion Coin Change Binary to Gray code using recursion Delete a linked list using recursion Product of 2 Numbers using Recursion Programs for Printing Pyramid Patterns using Recursion Length of longest palindromic sub-string : Recursion Program for Tower of Hanoi Algorithm Time Complexity Analysis | Tower Of Hanoi (Recursion) Program to calculate value of nCr using Recursion Find geometric sum of the series using recursion Convert a String to an Integer using Recursion DFS traversal of a Tree Bottom View of a Binary Tree using Recursion Write a program to print all Permutations of given String Print all subsets of a given Set or Array Print all possible paths from top left to bottom right of a mXn matrix Print all combinations of balanced parentheses Longest Common Subsequence (LCS) Hard Problems Find the value of a number raised to its reverse How to Sort a Stack using Recursion Reverse a Doubly linked list using recursion Given a string, print all possible palindromic partitions Check if a string is a scrambled form of another string Word Break Problem | DP-32 Print all palindromic partitions of a string N Queen Problem | Backtracking-3 Algorithm to Solve Sudoku | Sudoku Solver The Knight's tour problem Recursion-Algorithm.png Practice Sets on Recursion Recursive Practice Problems with Solutions Practice Questions for Recursion | Set 1 Practice Questions for Recursion | Set 2 Practice Questions for Recursion | Set 3 Practice Questions for Recursion | Set 4 Practice Questions for Recursion | Set 5 Practice Questions for Recursion | Set 6 Practice Questions for Recursion | Set 7 Practice questions for Linked List and Recursion Quiz based on Recursion: Top MCQs on Recursion Algorithm with Answers What is a Recursive Algorithm? A recursive algorithm is an algorithm that uses recursion to solve a problem. Recursive algorithms typically have two parts: Base case: Which is a condition that stops the recursion. Recursive case: Which is a call to the function itself with a smaller version of the problem. Types of Recursion There are several different recursion types and terms. These include: Direct recursion: This is typified by the factorial implementation where the methods call itself. In-Direct recursion: This happens where one method, say method A, calls another method B, which then calls method A. This involves two or more methods that eventually create a circular call sequence. Head recursion: The recursive call is made at the beginning of the method. Tail recursion: The recursive call is the last statement. When to Use Recursion? Recursion is a powerful technique that can be used to solve a wide variety of problems. However, it is important to use recursion carefully, as it can lead to stack overflows if not used properly. Recursion should be used when: The problem can be broken down into smaller subproblems that can be solved recursively. The base case is easy to identify. The recursive calls are tail recursive. Examples of Recursion Here are some common examples of recursion: Example 1: Factorial: The factorial of a number n is the product of all the integers from 1 to n. The factorial of n can be defined recursively as: factorial(n) = n \* factorial(n-1) Example 2: Fibonacci sequence: The Fibonacci sequence is a sequence of numbers where each number is the sum of the two preceding numbers. The Fibonacci sequence can be defined recursively as: fib(n) = fib(n-1) + fib(n-2) Applications of Recursion Algorithms: Here are some common applications of recursion: Tree and Graph Traversal: Depth-first search (DFS) and breadth-first search (BFS) Dynamic Programming: Solving optimization problems by breaking them into smaller subproblems Divide-and-Conquer: Solving problems by dividing them into smaller parts, solving each part recursively, and combining the results Backtracking: Exploring all possible solutions to a problem by recursively trying different options Combinatorics: Counting or generating all possible combinations or permutations of a set Comment More info Campus Training Program Next Article Tree Data Structure course-img 1417k+ interested Geeks Data Structures and Algorithms - Self Paced [Online Course] Explore course-img 112k+ interested Geeks Data Structures & Algorithms in JavaScript - Self Paced Course Explore course-img 344k+ interested Geeks Data Structures & Algorithms in Python - Self Paced Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Thanks for sending the **Recursion** content, Manoj! Here’s a **clean and structured summary for your MAANG-level DSA notes**, tailored specifically for fast revision, interview prep, and clarity.

**📘 DSA MAANG Notes – Recursion**

**🔁 What is Recursion?**

Recursion is a **programming technique** where a function **calls itself** (directly or indirectly) to solve a problem by dividing it into **smaller subproblems**.

⚠️ A recursive function must have:

* **Base Case** – The terminating condition
* **Recursive Case** – The function calling itself with smaller input

**🔃 Recursion vs Iteration**

| **Feature** | **Recursion** | **Iteration** |
| --- | --- | --- |
| **Approach** | Top-down | Bottom-up |
| **Termination** | Base condition | Loop condition |
| **Memory Use** | More (stack used) | Less |
| **Speed** | Generally slower | Generally faster |
| **Use Cases** | Tree/graph, backtracking, DP | Repeated arithmetic or traversal |

**🧠 Types of Recursion**

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| **Direct** | Function calls itself directly | factorial(n) |
| **Indirect** | Function A calls B, which calls A | A → B → A cycle |
| **Tail** | Recursive call is the last statement | Optimizable |
| **Head** | Recursive call happens before any other operation | Process after call |
| **Tree** | Multiple recursive calls per function | Fibonacci |
| **Linear** | Single recursive call per function | Factorial |

**✅ When to Use Recursion?**

Use recursion when:

* The problem has an easily identifiable **base case**.
* The problem can be broken into **smaller subproblems**.
* It naturally fits into recursive structure (e.g., **DFS, backtracking, divide-and-conquer**).

**🛠️ Common Recursive Problems**

**🟢 Easy:**

* Print 1 to n / n to 1 without loop
* Reverse a string
* Length of string
* Sum of digits
* Fibonacci
* Factorial
* Palindrome check
* Array min/max

**🟡 Medium:**

* Remove all adjacent duplicates
* Reverse a queue
* Sort a queue
* Coin change
* Tower of Hanoi
* nCr calculation
* Permutations
* Subsets of array
* Balanced parentheses

**🔴 Hard:**

* N-Queens
* Sudoku Solver
* Knight’s tour
* Sort a Stack
* Scrambled string
* Palindromic partitions

**📐 Sample Recursive Functions**

**Factorial**

int factorial(int n) { if (n == 0 || n == 1) return 1; return n \* factorial(n - 1); }

**Fibonacci (Exponential time)**

int fib(int n) { if (n == 0) return 0; if (n == 1) return 1; return fib(n - 1) + fib(n - 2); }

**Reverse a String**

void reverse(string &s, int i) { if (i >= s.length() / 2) return; swap(s[i], s[s.length() - 1 - i]); reverse(s, i + 1); }

**🧠 Important Concepts**

**Tail Recursion:**

* Recursive call is the **last thing** in the function.
* Some compilers **optimize** this to prevent stack overflow.

**Recursion in Tree Traversals:**

* **DFS** (Preorder, Inorder, Postorder) are recursive.
* **Backtracking** problems like N-Queens and Subsets also use recursion.

**📈 Applications of Recursion**

| **Area** | **Use Case Example** |
| --- | --- |
| **Tree Traversal** | DFS, Pre/In/Post Order |
| **Graphs** | DFS (recursive), Connected components |
| **Backtracking** | N-Queens, Maze solving, Word Break |
| **DP (Top Down)** | Memoized recursion |
| **Divide & Conquer** | Merge Sort, Quick Sort |
| **Combinatorics** | Generating permutations/subsets |

**❗ Tips for Interviews (MAANG)**

* Always define the **base case first**.
* Use **dry run** on small input (like n = 3).
* For optimization, convert to **memoization** or **DP**.
* Avoid unnecessary recomputation → Use **caching (map/array)**.
* Ask yourself: *Can this be converted to iteration?* (Tail Recursion hint!)

Would you like me to now:

* Merge all cleaned notes (Array, Strings, Recursion, etc.) into a PDF/Notion?
* Create a 30-day LeetCode plan for mastering recursion?
* Help you implement visualization tools (Recursion Tree, Stack Traces)?

Let me know how you want to continue — you’re building an elite-level DSA arsenal!

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... M DSA Course DSA Interview Problems on Matrix Practice Matrix MCQs on Matrix Tutorial on Matrix Matrix Traversal Sorting in Matrix Matrix Rotation Transpose of Matrix Inverse of Matrix Determinant of Matrix Matrix Application Adjoint & Inverse Matrix Sparse Matrix Matrix Exponentiation Explore GfG Courses Share Your Experiences Matrix Data Structure Matrix or Grid or 2D Array - Complete Tutorial Row-wise vs column-wise traversal of matrix Applications of Matrices and Determinants Basic Operations on Matrix Easy problems on Matrix Intermediate problems on Matrix Hard problems on Matrix DSA to DevelopmentCourse Matrix Data Structure Last Updated : 27 Feb, 2025 Matrix Data Structure is a two-dimensional array arranged in rows and columns. It is commonly used to represent mathematical matrices and is fundamental in various fields like mathematics, computer graphics, and data processing. Matrices allow for efficient storage and manipulation of data in a structured format. Basics of Matrix Data Structure Representation and Basic Operations Row-wise vs column-wise traversal Basic Problems Search in a Matrix Addition of Matrices Subtraction of Matrices Multiplication of Matrices Sort the given matrix Search in a sorted matrix Traverse Matrix using Recursion Transpose of a matrix Determinant of a Matrix Adjoint and Inverse of a Matrix Easy Problems Valid Suduku Validate Tic Tac Toe The Celebrity Problem Boundary Elements Matrix Zig-Zag Rotate Matrix Elements by 1 Scalar multiplication of a matrix Matrix Spiral Max in Each Row Distinct Commons in Rows Unique in a matrix Shift matrix elements row-wise by k Swap major and minor diagonals Squares of Matrix Diagonal Elements Sum of middle row and column in Matrix Check idempotent matrix Check diagonal and scalar matrix Identity Matrix Mirror across diagonal Pascal's Triangle Toeplit Matrix Possible moves of knight Medium Problems Conway’s Game Of Life Rotate a matrix by 90 Rotate a matrix by 180 Check if all rows are circular rotations Largest ‘X' Bordered Submatrix Largest Square Submatrix with All 1s Os in a row and column wise sorted Queries in a Matrix Count Pairs With Sum from 2 Matrices Pairs sum such from different rows Find all permuted rows of a given row No of Transformations to make two Matrix Equal Inplace M x N size matrix transpose Minimum flip to make Binary Matrix symmetric Magic Square Commons in all Rows of a Row-wise Sorted Hard Problems Median in a Row-Wise Sorted A Boolean Matrix Question Matrix Chain Multiplication Max sum square sub-matrix of given size Largest rectangle sub-matrix with all 1s Construct Ancestor Matrix from a Binary Tree K’th element in spiral form of matrix Largest ‘+’ formed by all ones in a binary matrix Shortest path in a Binary Maze Minimum Initial Points to Reach Destination Strassen’s Matrix Multiplication Max sum rectangle in a 2D matrix Sudoku Generator Matrix-Data-Structure Quick Links: ‘Practice Problems’ on Matrix ‘Quizzes’ on Matrix ‘Video Tutorials’ on Matrix Recommended: Learn Data Structure and Algorithms | DSA Tutorial Comment More info Campus Training Program Next Article Matrix or Grid or 2D Array - Complete Tutorial course-img 344k+ interested Geeks Data Structures & Algorithms in Python - Self Paced Explore course-img 797 interested Geeks GATE CSE + DA Complete Course Combo [2028] Explore course-img 1k+ interested Geeks GATE CSE + DA Complete Course Combo [2027] Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Thanks, Manoj! Here’s a **clean, structured, and MAANG-focused summary** of the **Matrix (2D Array / Grid) Data Structure** for your DSA notes. This will help you revise core concepts quickly and effectively tackle coding interviews.

**📘 DSA MAANG Notes – Matrix / 2D Array / Grid**

**🧱 What is a Matrix?**

A **Matrix** is a **2D array** of elements arranged in **rows and columns**.

int matrix[3][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };

* Matrix of size m x n has m rows and n columns.
* Widely used in: **image processing, graphs, dynamic programming, simulations**, etc.

**🛠️ Common Matrix Operations**

| **Operation** | **Description** |
| --- | --- |
| **Traversal** | Visiting each element row-wise / column-wise / spiral |
| **Transpose** | Swap matrix[i][j] with matrix[j][i] |
| **Matrix Addition** | C[i][j] = A[i][j] + B[i][j] |
| **Matrix Multiplication** | C[i][j] = sum(A[i][k] \* B[k][j]) |
| **Scalar Multiplication** | Multiply each element by a constant |
| **Matrix Rotation** | Rotate 90°, 180°, 270° (clockwise/anticlockwise) |
| **Matrix Exponentiation** | Repeated multiplication of square matrix |

**🔁 Traversal Patterns**

| **Pattern** | **Use Case Example** |
| --- | --- |
| Row-wise | for i in rows → for j in cols |
| Column-wise | for j in cols → for i in rows |
| Diagonal | Often in DP problems |
| Spiral | Matrix Spiral Traversal |
| Zig-Zag | Snake pattern |
| Boundary | Printing matrix boundary elements |

**🧠 Applications of Matrix**

* **Graphs (Adjacency Matrix)**
* **Game Development (Grids)**
* **Image processing (Pixels)**
* **Dynamic Programming (DP tables)**
* **Simulations (life, automata)**

**✅ Easy Interview Problems**

* Matrix Traversal
* Transpose a Matrix
* Spiral Matrix
* Diagonal sum
* Rotate Matrix by 90°
* Max in each row
* Search in sorted matrix (binary search on matrix)
* Pascal’s Triangle
* Valid Sudoku
* Boundary Elements

**🟡 Medium-Level Problems**

* Rotate matrix by 180°
* Game of Life
* Check all rows are circular rotations
* In-place matrix transpose (M x N)
* Count pairs with sum from 2 matrices
* Commons in all rows of row-wise sorted matrix
* Largest square sub-matrix with all 1s
* Minimum flips to make matrix symmetric
* Magic Square

**🔴 Hard-Level Problems**

* Matrix Chain Multiplication (DP)
* Maximum sum rectangle
* Strassen’s Matrix Multiplication (Divide & Conquer)
* Shortest path in Binary Maze (BFS)
* Sudoku Generator (Backtracking)
* Knight’s Tour
* Boolean Matrix Problem
* Largest rectangle of 1s
* Minimum initial points to reach destination
* Construct ancestor matrix from binary tree

**📐 Special Types of Matrices**

| **Type** | **Description** |
| --- | --- |
| **Identity Matrix** | Diagonal = 1, rest = 0 |
| **Diagonal Matrix** | Non-diagonal = 0 |
| **Scalar Matrix** | Diagonal same non-zero constant |
| **Symmetric Matrix** | A[i][j] == A[j][i] |
| **Sparse Matrix** | Most elements are 0 |
| **Toeplitz Matrix** | A[i][j] == A[i-1][j-1] |
| **Idempotent Matrix** | A \* A = A |

**🚀 Must Know Tricks for Interviews**

* Use binary search in **row-wise and column-wise sorted matrix**.
* **Spiral Traversal** often appears in 90% of matrix-based questions.
* Practice **rotating matrix in-place** (no extra space).
* Use **BFS** for shortest path questions in 0/1 grid.
* For **dynamic size sub-matrix** problems, prefix sum or DP is powerful.
* Learn **Matrix Chain Multiplication** via recursion + DP memoization.

**🧪 Sample Problem: Spiral Traversal**

vector<int> spiralOrder(vector<vector<int>>& matrix) { vector<int> res; if (matrix.empty()) return res; int top = 0, bottom = matrix.size() - 1; int left = 0, right = matrix[0].size() - 1; while (top <= bottom && left <= right) { for (int j = left; j <= right; ++j) res.push\_back(matrix[top][j]); top++; for (int i = top; i <= bottom; ++i) res.push\_back(matrix[i][right]); right--; if (top <= bottom) { for (int j = right; j >= left; --j) res.push\_back(matrix[bottom][j]); bottom--; } if (left <= right) { for (int i = bottom; i >= top; --i) res.push\_back(matrix[i][left]); left++; } } return res; }

Would you like me to now:

* Merge **Hashing**, **Strings**, **Recursion**, and **Matrix** notes into a PDF or Markdown?
* Make a **cheat sheet page** for each topic?
* Create **a practice list of top 50 matrix problems** from LeetCode & GFG?

Let me know how you'd like to proceed next — you're doing excellent work, and this is the exact level MAANG looks for!

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... 17 M DSA Course DSA Interview Problems on Linked List Practice Linked List MCQs on Linked List Linked List Tutorial Types of Linked List Singly Linked List Doubly Linked List Circular Linked List Circular Doubly Linked List Linked List vs Array Time & Space Complexity Advantages & Disadvantages Explore GfG Courses Share Your Experiences DSA Tutorial - Learn Data Structures and Algorithms Basics & Prerequisites Data Structures Getting Started with Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Advanced Interview Preparation Practice Problem DSA to DevelopmentCourse Linked List Data Structure Last Updated : 29 Jun, 2025 A linked list is a fundamental data structure in computer science. It mainly allows efficient insertion and deletion operations compared to arrays. Like arrays, it is also used to implement other data structures like stack, queue and deque. Here’s the comparison of Linked List vs Arrays Linked-list Linked List: Data Structure: Non-contiguous Memory Allocation: Typically allocated one by one to individual elements Insertion/Deletion: Efficient Access: Sequential Array: Data Structure: Contiguous Memory Allocation: Typically allocated to the whole array Insertion/Deletion: Inefficient Access: Random Basics of Linked List: Singly Linked List Doubly Linked List Circular Linked List Applications and Advantages of Linked List Operations of Linked Lists: Length of Linked List Print Linked List Linked List Insertion Search in a Linked List Deleting a given key Deleting at given position Delete a Linked List Nth Node from Start Nth Node from End Size of Doubly Linked List Easy Problems on Linked List: Remove every k-th node Middle of a Linked List Count Occurrences in a Linked List Circular Linked List Traversal Check if Circular Count Nodes in Circular List Deletion from a Circular Linked List Singly to circular Conversion Exchange first and last nodes in Circular Delete in a Doubly Linked List Reverse a Singly Linked List Reverse a Doubly Linked List Medium Problems on Linked List: Swap Nodes in Pairs Detect loop in a linked list Length of loop in linked list Design Browser History Remove duplicates from a sorted linked list Remove Duplicates from an Unsorted Linked List Intersection of two Sorted Linked Lists Partition a List QuickSort on Singly Linked List Split a Circular Linked List into two halves Merge Two Sorted Linked Lists Union and Intersection Merge Sort for Doubly Linked List Pairs with Sum in doubly linked list Insert in sorted way in doubly linked list Remove duplicates from an unsorted DLL Rotate a Linked List Rotate Doubly linked list by N nodes Delete a node with only its pointer given Segregate even and odd nodes Hard Problems on Linked List: Merge K Sorted Lists Intersection point of two Linked Lists. Implement LRU Cache Josephus Circle using circular linked list The Great Tree-List Recursion Problem. Clone a linked list with random pointer Binary Tree to Doubly Linked List | Set Reverse a Singly Linked List in Groups Reverse a Doubly Linked List in Groups Sublist Search (Search a linked list in another list) Construct a linked list from 2D matrix Rotate Linked List block wise Multiply two numbers represented by Lists Delete N nodes after M nodes Quick Links : ‘Practice Problems’ on Linked List ‘Videos’ on Linked List ‘Quizzes’ on Linked List Recommended: Learn Data Structure and Algorithms | DSA Tutorial pointer Comment More info Campus Training Program Next Article Stack Data Structure course-img 1417k+ interested Geeks Data Structures and Algorithms - Self Paced [Online Course] Explore course-img 112k+ interested Geeks Data Structures & Algorithms in JavaScript - Self Paced Course Explore course-img 344k+ interested Geeks Data Structures & Algorithms in Python - Self Paced Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Thanks, Manoj! Here's a clean, **interview-focused summary of Linked List** data structure for your MAANG DSA notes:

**📘 DSA MAANG Notes – Linked List**

**🔍 What is a Linked List?**

A **Linked List** is a **linear data structure** where each element (node) contains:

* **Data** (value)
* A **pointer** (reference) to the next node

Unlike arrays, linked lists **don’t require contiguous memory**.

**📊 Linked List vs Array**

| **Feature** | **Linked List** | **Array** |
| --- | --- | --- |
| **Memory** | Dynamic (non-contiguous) | Static or dynamic (contiguous) |
| **Access** | Sequential | Random |
| **Insertion/Deletion** | Efficient (O(1) at head) | Costly (O(n)) |
| **Search** | O(n) | O(1) for index-based |
| **Memory Usage** | Extra memory for pointer | Less overhead |

**📌 Types of Linked List**

| **Type** | **Description** |
| --- | --- |
| **Singly Linked List** | Points only to the next node |
| **Doubly Linked List** | Each node has prev and next pointers |
| **Circular Linked List** | Last node connects back to the head |
| **Circular DLL** | Combination of doubly and circular |

**⚙️ Common Operations**

| **Operation** | **Time Complexity** |
| --- | --- |
| **Insert at Head** | O(1) |
| **Insert at Tail** | O(n) (SLL), O(1) (DLL if tail pointer used) |
| **Search** | O(n) |
| **Delete a node** | O(n) |
| **Reverse a list** | O(n) |
| **Find middle element** | O(n) |
| **Detect cycle (Floyd’s)** | O(n) |

**🧪 Easy Interview Problems**

* Reverse a Linked List
* Middle of a Linked List
* Count Occurrences of a key
* Delete Nth node from end (two-pointer)
* Remove duplicates (sorted)
* Circular Linked List check
* Delete a Linked List
* Exchange first and last nodes (circular)

**🟡 Medium-Level Problems**

* Detect loop using Floyd's Cycle
* Length of loop
* Merge two sorted linked lists
* Rotate Linked List
* Segregate even and odd nodes
* Remove duplicates (unsorted)
* Browser History (DLL simulation)
* Insert in sorted DLL
* Delete a node (only pointer given)
* Intersection of two Linked Lists
* Partition a List
* QuickSort on Singly Linked List

**🔴 Hard-Level Problems**

* Merge K Sorted Lists (Min-Heap)
* Clone Linked List with Random Pointer (HashMap / O(1) trick)
* LRU Cache (DLL + HashMap)
* Josephus Circle
* Linked List to Binary Tree
* Reverse in Groups of K nodes
* Multiply 2 numbers represented as Linked Lists
* Sublist search (linked list in another list)
* Block-wise rotation of Linked List
* Construct linked list from 2D Matrix

**🧠 MAANG Insights**

✅ **Two-pointer technique**: Used in fast-slow pointer, cycle detection, kth from end.

✅ **Dummy Node Trick**: Simplifies edge cases for head insertions or merges.

✅ **Floyd’s Cycle Detection**:

bool hasCycle(ListNode \*head) { ListNode \*slow = head, \*fast = head; while (fast && fast->next) { slow = slow->next; fast = fast->next->next; if (slow == fast) return true; } return false; }

✅ **Reverse a Linked List** (Iterative):

ListNode\* reverseList(ListNode\* head) { ListNode\* prev = nullptr; while (head) { ListNode\* temp = head->next; head->next = prev; prev = head; head = temp; } return prev; }

✅ **Merge Two Sorted Lists**:

ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2) { ListNode dummy(0); ListNode\* tail = &dummy; while (l1 && l2) { if (l1->val < l2->val) tail->next = l1, l1 = l1->next; else tail->next = l2, l2 = l2->next; tail = tail->next; } tail->next = l1 ? l1 : l2; return dummy.next; }

✅ **Find Intersection Point**:

ListNode\* getIntersectionNode(ListNode \*headA, ListNode \*headB) { ListNode \*a = headA, \*b = headB; while (a != b) { a = a ? a->next : headB; b = b ? b->next : headA; } return a; }

**📘 Summary: When to Use Linked List?**

* When **insertions/deletions** are frequent
* When memory is **fragmented** (not contiguous)
* When implementing **stack, queue, graph adjacency list**
* When **dynamic size** is needed without resizing

Would you like me to now:

* Convert this and previous notes (Matrix, Recursion, Strings, Hashing) into a single PDF or Notion format?
* Create visual diagrams for each concept?
* Suggest a 10-day revision plan with coding practice?

Let me know how you want to proceed, you’re building a rock-solid DSA foundation!

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... M DSA Course DSA Interview Problems on Stack Practice Stack MCQs on Stack Stack Tutorial Stack Operations Stack Implementations Monotonic Stack Infix to Postfix Prefix to Postfix Prefix to Infix Advantages & Disadvantages ▲ Explore GfG Courses Share Your Experiences DSA Tutorial - Learn Data Structures and Algorithms Basics & Prerequisites Data Structures Getting Started with Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Advanced Interview Preparation Practice Problem DSA to DevelopmentCourse Stack Data Structure Last Updated : 27 Mar, 2025 A Stack is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). LIFO implies that the element that is inserted last, comes out first and FILO implies that the element that is inserted first, comes out last. It behaves like a stack of plates, where the last plate added is the first one to be removed. Think of it this way: Pushing an element onto the stack is like adding a new plate on top. Popping an element removes the top plate from the stack. Basics of Stack Data Structure Introduction to Stack Stack Array Implementation Stack Linked List Implementation Stack Implementation using Deque Applications of Stack Implementations of Stack in Different Languages Stack in C++ STL Stack in Java Stack in Python Stack in C# Stack in JavaScript Easy Problems on Stack Data Structures The Celebrity Problem Implement Queue using Stacks Implement two stacks in an array Implement Stack using Queues Stack using priority queue or heap Stack using single queue Infix to Postfix Prefix to Infix Prefix to Postfix Postfix to Prefix Postfix to Infix Infix To Prefix Check for balanced parentheses Arithmetic Expression Evalution Evaluation of Postfix Expression Reverse a stack using recursion Reverse Words Reverse a string using stack Reversing a Queue Reversing the first K of a Queue A Data Structure with O(1) Operations Medium Problems on Stack Data Structures k Stacks in an Array Mergable Stack Previous Smaller Element Next Greater Element Stock Span Problem Buildings Facing Sun Next Smaller of next Greater in an array Next Greater Frequency Element Max product of indexes of greater on left and right Iterative Tower of Hanoi Sort a stack using a temporary stack Reverse a stack without using extra space in O(n) Delete middle of a stack Check if a queue can be sorted into another queue Check if an array is stack sortable Iterative Postorder Traversal | Set 1 (Using Two Stacks) Index of closing bracket for a given opening bracket Max Diff between nearest left and right smaller elements Delete consecutive same words in a sequence Hard Problems on Stack Data Structures Largest Rectangular Area in a Histogram Sum of Max of all Subarrays Max of Mins of every window size Design a stack that supports getMin() Design a stack with max frequency operations Print next greater number of Q queries Length of the longest valid substring Iterative Postorder Traversal | Set 2 (Using One Stack) Print ancestors of a given binary tree node without recursion Expression contains redundant bracket or not Find if an expression has duplicate parenthesis Iterative method to find ancestors in a binary tree Stack Permutations Remove brackets from an algebraic string containing + and – operators Range Queries for Longest Correct Bracket Subsequence Quick Links : ‘Practice Problems’ on Stack ‘Videos’ on Stack ‘Quizzes’ on Stack Recommended: Learn Data Structure and Algorithms | DSA Tutorial Stack in Scala Comment More info Campus Training Program Next Article Queue Data Structure course-img 344k+ interested Geeks Data Structures & Algorithms in Python - Self Paced Explore course-img 57k+ interested Geeks Data Science Training Program Explore course-img 200k+ interested Geeks C Programming Course Online - Learn C with Data Structures Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Great, Manoj! Here's your **MAANG-focused DSA note summary for the Stack Data Structure**. It's concise, concept-driven, and includes example-based problems to help with revision and interviews.

**📘 DSA MAANG Notes – Stack Data Structure**

**🔍 What is a Stack?**

A **Stack** is a **linear data structure** that follows:

* **LIFO** – Last In, First Out
* Operations:
  + **Push** → insert at top
  + **Pop** → remove from top
  + **Peek/Top** → view top element without removing
  + **isEmpty** → check if stack is empty

🧠 Think of it like a pile of plates or a browser's back button history.

**📊 Stack – Key Characteristics**

| **Feature** | **Details** |
| --- | --- |
| Order | LIFO / FILO |
| Insert/Remove | At one end only (top) |
| Access | Only top accessible directly |
| Common Use Cases | Expression parsing, Backtracking, DFS, Undo features |

**⚙️ Stack Implementations**

| **Method** | **Description** |
| --- | --- |
| **Array-based Stack** | Fixed size, fast but limited |
| **Linked List Stack** | Dynamic size |
| **Deque (STL/Java)** | Efficient insert/remove at both ends |

**✅ Applications of Stack**

* Expression evaluation (Postfix, Prefix, Infix)
* Parentheses checking
* Backtracking (Maze, Sudoku)
* Undo features (Text Editors)
* Reversing (Strings, Data)
* DFS traversal in Trees/Graphs
* Browser history or navigation stack

**🔡 Expression Conversions (Very Important)**

| **From → To** | **Description** |
| --- | --- |
| Infix → Postfix | Use stack (Shunting Yard Algorithm) |
| Prefix → Postfix | Reverse + convert + reverse logic |
| Postfix → Infix | Use operand stack |
| Balanced Parentheses | Stack-based parsing |

**🧪 Easy Stack Interview Problems**

* Reverse a Stack using Recursion
* Balanced Parentheses Checker
* Implement Queue using 2 Stacks
* Implement Stack using Queues
* Infix to Postfix Conversion
* Evaluate Postfix Expression
* Reverse Words / Strings
* Celebrity Problem (O(n) with stack)

**🟡 Medium-Level Problems**

* Next Greater Element
* Previous Smaller Element
* Stock Span Problem
* Buildings Facing Sun
* Sort a Stack using another Stack
* Delete Middle Element of Stack
* Check Stack Sortable Array
* Reverse a stack without extra space

**🔴 Hard-Level Problems**

* Largest Rectangle in Histogram  
  🔁 Uses **monotonic stack** for height-based area calc
* Maximum of Minimums of All Window Sizes
* Stack with getMin() in O(1)
* Longest Valid Parentheses Substring
* Expression with Redundant Brackets
* Stack Permutations (Valid or Not)
* Design Stack with Max Frequency (Leetcode: FreqStack)
* Print Ancestors without recursion (Binary Tree)

**🔄 Monotonic Stack Use Cases**

| **Pattern** | **Application** |
| --- | --- |
| **Next Greater Element** | Stock Span, Histogram Max Area |
| **Previous Smaller Element** | Max of Mins in window, Histogram |
| **Next Smaller Element** | Building View, Rainwater Trapping |

**🔧 Common Stack Code Snippets**

**✅ Reverse a Stack (using Recursion)**

void insertAtBottom(stack<int>& s, int val) { if (s.empty()) { s.push(val); return; } int top = s.top(); s.pop(); insertAtBottom(s, val); s.push(top); } void reverseStack(stack<int>& s) { if (s.empty()) return; int top = s.top(); s.pop(); reverseStack(s); insertAtBottom(s, top); }

**✅ Largest Rectangle in Histogram**

int largestRectangleArea(vector<int>& heights) { stack<int> s; int maxArea = 0; heights.push\_back(0); *// Sentinel* for (int i = 0; i < heights.size(); i++) { while (!s.empty() && heights[s.top()] > heights[i]) { int h = heights[s.top()]; s.pop(); int width = s.empty() ? i : i - s.top() - 1; maxArea = max(maxArea, h \* width); } s.push(i); } return maxArea; }

**✅ Get Min in O(1)**

stack<int> s, minS; void push(int val) { s.push(val); if (minS.empty() || val <= minS.top()) minS.push(val); } void pop() { if (s.top() == minS.top()) minS.pop(); s.pop(); } int getMin() { return minS.top(); }

**💼 MAANG Interview Takeaways**

✅ Focus on:

* Expression problems
* Stack design patterns (Monotonic Stack, Frequency Stack)
* Recursion with Stack
* Real-world stack usage like undo operations, path finding, etc.

Would you like me to combine all topics into a printable and sharable **DSA Handbook PDF** or an **interactive Notion workspace** for daily revision?

Also, let me know when you're ready to move on to **Queue, Trees, or Graphs**. You're building excellent mastery with these structured notes!

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... M DSA Course DSA Interview Problems on Queue Practice Queue MCQs on Queue Queue Tutorial Operations Applications Implementation Stack vs Queue Types of Queue Circular Queue Deque Priority Queue Stack using Queue Advantages & Disadvantages Explore GfG Courses Share Your Experiences DSA Tutorial - Learn Data Structures and Algorithms Basics & Prerequisites Data Structures Getting Started with Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Advanced Interview Preparation Practice Problem DSA to DevelopmentCourse Queue Data Structure Last Updated : 20 Jun, 2025 A Queue Data Structure is a fundamental concept in computer science used for storing and managing data in a specific order. It follows the principle of "First in, First out" (FIFO), where the first element added to the queue is the first one to be removed. It is used as a buffer in computer systems where we have speed mismatch between two devices that communicate with each other. For example, CPU and keyboard and two devices in a network Queue is also used in Operating System algorithms like CPU Scheduling and Memory Management, and many standard algorithms like Breadth First Search of Graph, Level Order Traversal of a Tree. Basics Introduction Basic Operations Array Implementations Linked List Implementation Applications Implementations in various Programming Languages Queue in C++ STL Queue In Java Queue In Python Queue In C# Queue in JavaScript Queue in Go Language Queue in Scala Implementing Other Data Structures Using Queue Implement a stack using single queue Implement Queue using Stacks LRU Cache Implementation Easy Problems on Queue Implement Stack using Queues Minimum Depth of a Binary Tree BFS for a Graph Detect cycle in an undirected graph using BFS Right View of a Binary Tree Check whether a graph is Bipartite Medium Problems on Queue Implement k Queues in a single array Flatten a multilevel linked list Level with maximum number of nodes Find if there is a path between two vertices in a directed graph Print all nodes between two given levels in Binary Tree Find next right node of a given key Minimum steps to reach target by a Knight Islands in a graph using BFS Find the first non-repeating in a stream Hard Problems on Queue Sliding Window Maximum (Maximum of all subarrays of size K) Flood Fill Algorithm Minimum time required to rot all oranges Shortest path in a Binary Maze An Interesting Method to Generate Binary Numbers from 1 to n Maximum cost path from source node to destination Shortest distance between two cells in a matrix or grid Snake and Ladder Problem Find shortest safe route in a path with landmines Count all possible walks from a source to a destination with exactly K edges Minimum Cost of Simple Path between two nodes in a directed and weighted graph Minimum Cost Path in a directed graph via given set of intermediate nodes Find the first circular tour that visits all petrol pumps Quick Links: ‘Practice Problems’ on Queue ‘Quizzes’ on Queue Learn Data Structure and Algorithms | DSA Tutorial Comment More info Campus Training Program Next Article Tree Data Structure course-img 344k+ interested Geeks Data Structures & Algorithms in Python - Self Paced Explore course-img 112k+ interested Geeks Data Structures & Algorithms in JavaScript - Self Paced Course Explore course-img 9k+ interested Geeks GATE CS/IT 2026 Complete Course [with Placement Preparation] Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Awesome, Manoj! Here's your **MAANG-ready summary of Queue Data Structure** with key concepts, patterns, and most frequently asked problems in interviews.

**📘 DSA MAANG Notes – Queue Data Structure**

**🔍 What is a Queue?**

A **Queue** is a linear data structure that follows:

* **FIFO** – First In First Out
* Element inserted first is removed first
* Works like a queue at a ticket counter or printer buffer

**📊 Characteristics of Queue**

| **Property** | **Details** |
| --- | --- |
| Insertion | Rear end |
| Deletion | Front end |
| Access | Sequential (No random access) |
| Time Complexity | Enqueue – O(1), Dequeue – O(1) |
| Real-life Usage | CPU scheduling, OS task management, BFS, cache |

**⚙️ Queue Types**

| **Type** | **Description** |
| --- | --- |
| **Simple Queue** | Standard FIFO queue |
| **Circular Queue** | Reuses freed spaces |
| **Deque** (Double Ended Queue) | Insert/delete from both ends |
| **Priority Queue** | Elements removed by priority not position |

**✅ Queue Operations**

* enqueue() – Add to rear
* dequeue() – Remove from front
* peek() – View front
* isEmpty() – Check if empty
* isFull() – Check if full (for array)

**🔨 Queue Implementations**

| **Method** | **Pros/Cons** |
| --- | --- |
| **Array** | Fixed size |
| **Linked List** | Dynamic size |
| **Deque/Library** | Easy with built-in STL (C++: queue, Java: Queue, Python: collections.deque) |

**🔁 Applications of Queue**

* CPU scheduling
* I/O buffers
* BFS traversal
* Level-order traversal in trees
* Handling requests (printer, server)

**🔄 Implementing Other DS Using Queue**

* **Stack using 2 Queues**
* **Queue using 2 Stacks**
* **LRU Cache** using Queue + HashMap

**🧪 Easy Queue Problems**

* Implement Stack using Queue
* Implement Queue using Stack
* BFS of Graph
* Right View of Binary Tree (level-order logic)
* Detect Cycle in Undirected Graph (using BFS)
* First Non-Repeating Character in a Stream
* Minimum Depth of Binary Tree

**🟡 Medium-Level Problems**

* Level with Max Nodes (Tree)
* Flatten a Multilevel Linked List
* Knight Moves in Chess Board (shortest path – BFS)
* Path Exists between Nodes in Graph (BFS)
* Nodes between two levels in Binary Tree
* Circular Queue implementation
* K Queues in One Array

**🔴 Hard-Level Problems**

* **Sliding Window Maximum**  
  ⏱️ O(n) using deque  
  🧠 Very popular in interviews (Amazon, Google)
* **Rotting Oranges**  
  🍊 Multi-source BFS
* **Shortest Path in Binary Maze**  
  2D BFS for pathfinding
* **Snake and Ladder Problem**  
  BFS-based minimum dice throws
* **Generate Binary Numbers from 1 to n**  
  Output in binary using queue
* **Find Shortest Safe Path in Landmines Grid**
* **Minimum Cost/Steps to Reach Destination in Grid**

**💡 MAANG Insights**

| **Pattern** | **Tip** |
| --- | --- |
| **Multi-source BFS** | For problems like Rotting Oranges, Flood Fill |
| **0-1 BFS** | Use deque for weighted edges of 0 and 1 |
| **Level-order pattern** | Many tree-based questions use it |
| **Sliding Window** | Master deque-based window logic |

**💻 Code Snippet: BFS Template**

void BFS(int start, vector<vector<int>>& adj, vector<bool>& visited) { queue<int> q; q.push(start); visited[start] = true; while (!q.empty()) { int node = q.front(); q.pop(); cout << node << " "; for (int neighbor : adj[node]) { if (!visited[neighbor]) { visited[neighbor] = true; q.push(neighbor); } } } }

**🧠 Interview-Style Problem: First Non-Repeating Character in Stream**

string FirstNonRepeating(string A) { queue<char> q; vector<int> count(26, 0); string ans = ""; for (char c : A) { q.push(c); count[c - 'a']++; while (!q.empty() && count[q.front() - 'a'] > 1) { q.pop(); } ans += q.empty() ? '#' : q.front(); } return ans; }

Would you like me to create:

✅ A PDF summary for Arrays, Hashing, Matrix, Linked List, Stack, Queue  
✅ A Notion template for daily review + Leetcode practice tracker  
✅ Or continue to **Tree Data Structures** next?

Let me know what works best for your study workflow, Manoj!

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... DSA Course DSA Interview Problems on Queue Practice Queue MCQs on Queue Queue Tutorial Operations Applications Implementation Stack vs Queue Types of Queue Circular Queue Deque Priority Queue Stack using Queue Advantages & Disadvantages Explore GfG Courses Share Your Experiences Applications, Advantages and Disadvantages of Deque Introduction and Array Implementation of Deque Introduction to Divide and Conquer Algorithm Real-life Applications of Data Structures and Algorithms (DSA) Applications, Advantages and Disadvantages of Depth First Search (DFS) DSA to DevelopmentCourse Deque - Introduction and Applications Last Updated : 01 Jul, 2025 Deque or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends. Below is an example program of deque in different languages. Deque Deque can act as both Stack and Queue It is useful in many problems where we need to have a subset of all operations also like insert/remove at front and insert/remove at the end. It is typically implemented either using a doubly linked list or circular array. Implementations of Deque in Different Languages deque in C++ Deque in Java deque in Python Deque in JavaScript Below are example programs in different languages. Try it on GfG Practice redirect icon Loading Playground... #include <iostream> #include <deque> using namespace std; int main() { deque<int> dq; dq.push\_back(10); dq.push\_back(20); dq.push\_front(30); // Print deque elements for (int x : dq) cout << x << " "; cout << endl; // Pop from front and back dq.pop\_front(); dq.pop\_back(); // Print deque elements after pop for (int x : dq) cout << x << " "; return 0; } Deque Basics Array Implementation of Deque Linked List Implementation of Deque Practice Problems Based on Deque Basic Problems Difference between Queue and Deque Deque Implementation in Python First and Last Elements of Deque in Python Add Element at Front of a Deque Remove an Element from Front of Deque Minimize Maximum Difference Between Adjacent Elements Easy Problems Rearrange Linked List to Alternate First and Last Substring with Maximum Frequency Prefixes as Suffixes of a String Level order traversal in spiral form String after processing backspace characters Generate a Sequence by inserting positions Lexicographically largest permutation Check if Strings Can Be Made Equal Medium Problems Stack and Queue Using ArrayDeque in Java Implement Stack and Queue using Deque Generate Bitonic Sequence Rearrange Array Elements Longest Subarray with Absolute Difference ≤ X Reverse a Linked List in groups Max Sum Subsequence with K Distant Elements Nth term of given recurrence relation Max Subarray Length with K Increments Largest String after Deleting K Characters Segregate even and odd nodes in a Linked List Generate Permutation with Unique Adjacent Differences 0-1 BFS Min Deques to Sort Array Min Number by Applying + and \* Operations Comment More info Campus Training Program Next Article Deque - Introduction and Applications course-img 344k+ interested Geeks Data Structures & Algorithms in Python - Self Paced Explore course-img 112k+ interested Geeks Data Structures & Algorithms in JavaScript - Self Paced Course Explore course-img 200k+ interested Geeks C Programming Course Online - Learn C with Data Structures Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Excellent again, Manoj! Here is your **MAANG-focused Deque (Double-Ended Queue) DSA Summary**, optimized for interviews and technical assessments.

**📘 DSA MAANG Notes – Deque (Double-Ended Queue)**

**🔍 What is a Deque?**

A **Deque** (pronounced "deck") is a **double-ended queue**, where:

* You can **insert and delete from both ends**
* Combines features of both **Stack (LIFO)** and **Queue (FIFO)**

**📊 Characteristics of Deque**

| **Operation** | **Front** | **Rear** |
| --- | --- | --- |
| Insertion | ✅ | ✅ |
| Deletion | ✅ | ✅ |
| Implementation | Doubly Linked List / Circular Array |  |
| Time Complexity | O(1) for all operations in good implementations |  |

**⚙️ Types of Deques**

| **Type** | **Description** |
| --- | --- |
| **Input-Restricted** | Insertion at one end only, deletion at both |
| **Output-Restricted** | Deletion at one end only, insertion at both |

**🧰 Common Operations**

dq.push\_front(x); *// Insert at front* dq.push\_back(x); *// Insert at rear* dq.pop\_front(); *// Remove from front* dq.pop\_back(); *// Remove from rear* dq.front(); *// Peek front* dq.back(); *// Peek rear* dq.empty(); *// Check if empty* dq.size(); *// Current size*

**🛠️ Applications of Deque**

| **Application** | **Explanation** |
| --- | --- |
| **Sliding Window Maximum** | Maintain a max window using deque |
| **Palindrome Checker** | Compare both ends |
| **Undo/Redo functionality** | Front = undo, Rear = redo |
| **0-1 BFS** | Insert at front if cost = 0, back if cost = 1 |
| **Text editors, Media players** | Navigation buffers |
| **Level order traversal in Spiral form** | Alternate left-to-right and right-to-left |

**🔁 Deque Implementations**

| **Implementation Method** | **Usage** |
| --- | --- |
| **Doubly Linked List** | Flexible, dynamic memory |
| **Circular Array** | Fixed size but cache-friendly |
| **Language STL/Library** | deque in C++ STL, Python collections.deque, Java ArrayDeque |

**💻 C++ Example**

#include <iostream> #include <deque> using namespace std; int main() { deque<int> dq; dq.push\_back(10); dq.push\_front(20); dq.push\_back(30); for (int x : dq) cout << x << " "; *// Output: 20 10 30* dq.pop\_front(); *// Removes 20* dq.pop\_back(); *// Removes 30* cout << "\nAfter popping: "; for (int x : dq) cout << x << " "; *// Output: 10* return 0; }

**✅ Easy Problems**

* Rearrange linked list to alternate first and last
* Spiral level order traversal (zig-zag)
* String after processing backspace characters (simulate editor)
* First and last element tracking
* Generate sequences and permutations with constraints

**🟡 Medium Problems**

* Implement Stack & Queue using Deque
* Longest subarray with absolute diff ≤ X
* Bitonic Sequence Generator
* Max sum subsequence with k distant elements
* Reverse Linked List in groups
* Lexicographically largest permutations using deque logic
* Generate unique permutations with adjacent constraints

**🔴 Hard/Tricky Problems**

| **Problem** | **Notes** |
| --- | --- |
| **Sliding Window Maximum** | O(n) time using Deque (must-know for MAANG) |
| **0-1 BFS** | Graph traversal with weighted edges 0/1 |
| **Minimum Deques to Sort Array** | Greedy + Deque strategy |
| **Largest String after K deletions** | Deque + Heap |
| **Max subarray with k Increments** | Greedy window strategy using deque |
| **Minimize Max Adjacent Difference** | Sort + Greedy + Deque |

**🧠 Patterns To Master**

| **Pattern** | **Problem Type** |
| --- | --- |
| **Sliding Window (Fixed Size)** | Max sum, max/min element, average |
| **Two-Ended BFS (0-1 BFS)** | Graph path problems |
| **Greedy using Deque** | Lexicographic manipulations, rearrangements |
| **Deque for Group Reversals** | Reverse k-groups in linked list |

**💡 MAANG Interview Tip**

Deques are rarely asked directly in interviews, but the logic **behind** deques powers **many key problems** like sliding window, BFS variations, spiral traversals, etc.  
You **must master their use** in:

* BFS variations
* Stack & Queue hybrid problems
* Maximum of all subarrays of size k (most famous)

Would you like me to continue the same style for **Trees** next?

Also let me know if you'd like a compiled **PDF of all your notes so far** (Matrix, Linked List, Stack, Queue, Deque) – I can arrange it cleanly with visual hierarchy and practice problems included.

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... M DSA Course DSA Interview Problems on Tree Practice Tree MCQs on Tree Tutorial on Tree Types of Trees Basic operations Tree Traversal Binary Tree Complete Binary Tree Ternary Tree Binary Search Tree Red-Black Tree AVL Tree Full Binary Tree B-Tree Advantages & Disadvantages Explore GfG Courses Share Your Experiences DSA Tutorial - Learn Data Structures and Algorithms Basics & Prerequisites Data Structures Getting Started with Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Advanced Interview Preparation Practice Problem DSA to DevelopmentCourse Tree Data Structure Last Updated : 12 May, 2025 Tree Data Structure is a non-linear data structure in which a collection of elements known as nodes are connected to each other via edges such that there exists exactly one path between any two nodes. tree-data-structure-banners-(2) Basics of Tree Data Structure Introduction to Tree Types of Trees in Data Structures Applications of tree data structure Binary Tree Binary Tree Guide Applications and Advantages of Binary Tree Properties of Binary Tree Tyes of Binary Tree Inorder Traversal of Binary Tree Preorder Traversal of Binary Tree Postorder Traversal of Binary Tree Level Order Tree Traversal Height or Depth of a Binary Tree Height and Depth of Node Level of a Given Node in Tree Search a Node in Binary Tree Find the Parent in Binary Tree Insertion in a Binary Tree Deletion in a Binary Tree Enumeration of Binary Trees Find all Leaf nodes Binary Tree (Array implementation) Binary Search Tree Binary Search Tree Guide Applications of BST Insertion in Binary Search Tree Searching in Binary Search Tree Binary Search Tree (BST) Traversals – Inorder, Preorder, Post Order Deletion in Binary Search Tree More Problems on Binary Search Tree AVL Tree AVL Tree Data Structure Insertion in an AVL Tree Deletion in an AVL Tree Weak AVL or Rank Balanced Trees Insertion, Searching and Deletion in AVL trees containing a parent node pointer AVL with duplicate keys Count greater nodes in AVL tree How to insert Strings into an AVL Tree Minimum number of nodes in an AVL Tree with given height Optimal sequence for AVL tree insertion (without any rotations) Different shapes of AVL possible at height h Red-Black Tree Introduction to Red-Black Tree Insertion in Red-Black Tree Red-Black Trees | Top-Down Insertion Deletion in Red-Black Tree Applications, Advantages, and Disadvantages of Red-Black Tree Ternary Search Tree Ternary Search Tree Ternary Search Tree meaning & definition in DSA Ternary Search Tree (Deletion) How to implement text Auto-complete feature using Ternary Search Tree Longest word in ternary search tree n-ary or Generic Tree Generic Trees(N-ary Trees) What is Generic Tree or N-ary Tree Depth of an N-ary Tree Children of a Node in an N-ary Tree Find Siblings in an N-ary Tree Mirror of n-ary Tree Diameter of an N-ary Tree Level Order Traversal of N-ary Tree Sum of all elements of N-ary Tree Serialize and Deserialize an N-ary Tree B Tree Introduction of B-Tree What is B-Tree? | B-Tree meaning Insert Operation in B-Tree Delete Operation in B-Tree B-Tree Insert without aggressive splitting B+ Tree Introduction of B+ Tree What is B+ Tree | B+ Tree meaning Insertion in a B+ tree Deletion in B+ Tree Other types of Trees Ternary Tree: What is Ternary Tree Create a Doubly Linked List from a Ternary Tree Interval Tree: Interval Tree Interval Tree using GNU Tree-based container 2-3-4 Tree: 2-3-4 Tree 2-3 Trees | (Search, Insert, and Deletion) Trees vs other Data Structures Difference between graph and tree Comparison between Heap and Tree What is the difference between Heap and Red-Black Tree? Difference between Binary Search Tree and Binary Heap Difference between Stack and Tree Difference between an array and a tree Comparison among different Tree Data Structures Difference between General tree and Binary tree Difference between Binary Tree and Binary Search Tree Difference between Binary tree and B-tree Difference between B tree and B+ tree Difference between Full and Complete Binary Tree Difference between Binary Search Tree and AVL Tree Red Black Tree vs AVL Tree Problems based on Tree Data Structure Problems Difficulty Level Solve Height of Binary Tree Easy Solve Determine if two trees are identical Easy Solve Mirror tree Easy Solve Symmetric Tree Easy Solve Diameter of tree Easy Solve Checked for Balanced tree Easy Solve Children Sum Parent Easy Solve Check for BST Easy Solve Array to BST Easy Solve Largest value in each level of binary tree Easy Solve Maximum GCD of siblings of a binary tree Easy Solve Zigzag Tree Traversal Easy Solve Inorder Successor in BST Easy Solve Kth Largest Element in a BST Easy Solve Check if subtree Medium Solve Single Valued Subtree Medium Solve Unique BSTs Medium Solve Inorder Traversal (iterative) Medium Solve Preorder Traversal (iterative) Medium Solve Postorder Traversal(iterative) Medium Solve Vertical Traversal of a Binary Tree Medium Solve Boundary Traversal Medium Solve Construct Binary Tree from Parent array Medium Solve Construct Binary Tree from Preorder and Inorder Traversal Medium Solve Preorder Traversal and BST Medium Solve Construct tree from preorder traversal Medium Solve Minimum distance between two given nodes Medium Solve Maximum sum leaf to root path Medium Solve Odd Even Level Difference Medium Solve Lowest Common Ancestor of a Binary Tree Medium Solve Ancestors in Binary Tree Medium Solve Remove BST keys outside the given range Medium Solve Pair with given target in BST Medium Solve Sum Tree Medium Solve BST to greater sum tree Medium Solve BST to max heap Medium Solve Clone binary tree with random pointer Medium Solve Maximum sum of non adjacent nodes Medium Solve Largest BST in a Binary Tree Medium Solve Extreme nodes in alternate order Medium Solve Connect nodes at same level Hard Solve Nodes at given distance in a Binary Tree Hard Solve Sorted Linked List to BST Hard Solve Binary Tree to Doubly Linked List Hard Solve Maximum sum path between two leaf nodes Hard Solve K-Sum Paths Hard Solve Number of turns in a binary tree Hard Solve Merge two BST’s Hard Solve Fixing two nodes of a BST Hard Solve Burn Binary Tree Hard Solve Quick Links: Practice problems on Trees Quizzes on Tree Comment More info Campus Training Program Next Article Graph Data Structure course-img 1417k+ interested Geeks Data Structures and Algorithms - Self Paced [Online Course] Explore course-img 112k+ interested Geeks Data Structures & Algorithms in JavaScript - Self Paced Course Explore course-img 344k+ interested Geeks Data Structures & Algorithms in Python - Self Paced Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Excellent, Manoj! Here is a **MAANG-optimized summary of the Tree Data Structure**, based on the details you’ve shared from GeeksforGeeks, and shaped specifically for technical interviews and system understanding.

**🌲 DSA MAANG Notes – Tree Data Structures**

**🔍 What is a Tree?**

A **tree** is a non-linear data structure made of **nodes** connected by **edges**.

* One node is the **root**
* Each node may have **children**
* No **cycles** and exactly **one path** between any two nodes

**📘 Terminology**

| **Term** | **Meaning** |
| --- | --- |
| Root | Topmost node |
| Parent | Node with children |
| Child | Node derived from parent |
| Leaf | Node with no children |
| Height | Longest path from node to a leaf |
| Depth | Number of edges from root to the node |
| Degree | Number of children of a node |
| Subtree | Any node and its descendants |

**🧱 Types of Trees**

| **Type** | **Description** |
| --- | --- |
| **Binary Tree** | Each node has ≤ 2 children |
| **Full Binary Tree** | Every node has 0 or 2 children |
| **Complete Binary Tree** | All levels filled except possibly the last, filled left to right |
| **Perfect Binary Tree** | All internal nodes have 2 children and all leaves are at the same level |
| **Balanced Tree** | Height difference between left and right ≤ 1 |
| **Binary Search Tree (BST)** | Left < root < right |
| **AVL Tree** | Self-balancing BST |
| **Red-Black Tree** | BST with balancing rules (color-coded) |
| **Ternary Tree** | Node with 3 children (used in autocomplete) |
| **N-ary Tree** | Node with N children |
| **B-Tree / B+ Tree** | Used in databases, self-balancing with multiple children |

**🌲 Binary Tree Traversals**

| **Type** | **Order** |
| --- | --- |
| **Inorder** | Left → Root → Right |
| **Preorder** | Root → Left → Right |
| **Postorder** | Left → Right → Root |
| **Level Order** | Level by level (BFS) |

✅ **Iterative** and **Recursive** implementations are interview favorites.

**🧮 Operations on Trees**

| **Operation** | **Time Complexity** |
| --- | --- |
| Insertion (Binary Tree) | O(n) |
| Insertion (BST) | O(log n) average, O(n) worst |
| Search in BST | O(log n) average, O(n) worst |
| Deletion in BST | O(log n) |
| Traversals | O(n) |
| Height Calculation | O(n) |

**📚 Applications of Trees**

* **BST**: Efficient searching/sorting
* **AVL/Red-Black Tree**: Balanced search trees
* **Trie**: Dictionary, autocomplete
* **Segment Tree**: Range queries
* **N-ary Tree**: Hierarchical data (HTML/XML)
* **Parse Tree**: Compilers
* **B-Tree/B+ Tree**: Databases, file systems

**🧠 MAANG Favorite Tree Problems (by type)**

**🔹 Easy**

* Inorder, Preorder, Postorder traversal (rec & iter)
* Symmetric Tree
* Mirror Tree
* Height of Binary Tree
* Check for BST
* Array to BST
* Diameter of Tree
* Zigzag Tree Traversal

**🟡 Medium**

* Construct Tree from Inorder + Preorder
* Vertical Order Traversal
* Boundary Traversal
* LCA (Lowest Common Ancestor)
* Distance Between Two Nodes
* Maximum Sum Path from Root to Leaf
* Sum Tree, Ancestors, Remove nodes outside range (BST)

**🔴 Hard**

* K Sum Paths
* Burn Binary Tree
* Clone Binary Tree with Random Pointer
* Sorted Linked List to BST
* Max Sum Between Two Leaves
* Merge Two BSTs
* Number of Turns in Binary Tree
* Connect Nodes at Same Level (constant space)

**⚖️ BST vs AVL vs Red-Black Tree**

| **Feature** | **BST** | **AVL Tree** | **Red-Black Tree** |
| --- | --- | --- | --- |
| Balance | No | Strict | Relaxed |
| Rotations | No | More | Fewer |
| Insert/Search | O(n) worst | O(log n) | O(log n) |
| Use Case | Simple cases | Time-critical | Complex systems (OS, DBs) |

**💡 MAANG-Level Tips**

* **Understand all Tree Traversals (Recursive + Iterative)** – Most common.
* **Practice Tree Construction from Traversals** (Pre+In, Post+In).
* **Master LCA (for Binary Tree and BST separately)**.
* **Burn Tree, Clone Tree with Random Pointers, K-Sum Paths** are **excellent hard-level prep**.
* Understand **height-balanced tree rotations** for AVL and Red-Black Trees.

Would you like me to proceed to the **Graph** next in the same structured format?

And if you'd like, I can also create:

* A clean **PDF** of all DSA notes so far
* A Notion-compatible layout
* Flashcards for quick revision

Let me know how you want to use these!

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... DSA Course DSA Interview Problems on Heap Practice Heap MCQs on Heap Heap Tutorial Binary Heap Building Heap Binomial Heap Fibonacci Heap Heap Sort Heap vs Tree Leftist Heap K-ary Heap Advantages & Disadvantages Explore GfG Courses Share Your Experiences Heap Data Structure Introduction to Heap - Data Structure and Algorithm Tutorials Binary Heap Advantages and Disadvantages of Heap Time Complexity of building a heap Applications of Heap Data Structure Comparison between Heap and Tree When building a Heap, is the structure of Heap unique? Some other type of Heap Easy problems on Heap Medium problems on Heap Hard problems on Heap DSA to DevelopmentCourse Heap Data Structure Last Updated : 26 Apr, 2025 A Heap is a complete binary tree data structure that satisfies the heap property: for every node, the value of its children is greater than or equal to its own value. Heaps are usually used to implement priority queues, where the smallest (or largest) element is always at the root of the tree. min-heap-1.webpmin-heap-1.webp Basics Binary Heap Applications of Heap Data Structure Library Implementations Heap in C++ STL priority\_queue in C++ PriorityQueue in Java PriorityQueue in C# heapq in Python Heap in JavaScript Easy Problems Check if an array is Heap? K’th Smallest & K'th Largest Heap Sort Heap Sort for Decreasing Order Height of a Heap with N Nodes Medium Problems Nearly Sorted Array K Largest Elements & K Smallest k Closest Elements Max diff between two m Sized Subsets Check if Binary Tree is Heap Huffman Coding Nodes less than a value in a Min Heap. Connect n ropes with minimum cost Maximum distincts after removing k elements K maximum sum combinations from two arrays Median in a stream K’th largest in a stream Largest triplet product in a stream k most frequent Sort by Frequency Min Heap to Max Heap Check for Min-Heap from Level Order Hard Problems The Skyline Problem K-th Largest Sum Subarray Find K Closest Points to the Origin Merge k sorted arrays Merge K Sorted Linked Lists Range Covering K Sorted Lists Prim's Minimum Spanning Tree Dijkstra's Shortest Path Sort numbers stored on different machines Smallest Derangement of Sequence Largest Derangement of a Sequence Convert BST to Min Heap Merge two binary Max Heaps Minimum product of k integers Leaf starting point in a Binary Heap Rearrange characters such that no two same adjacent Sum of all between k1’th and k2’th smallest Min sum of two formed from given digits A data structure with min and max Other Types of Heaps Binomial Heap Fibonacci Heap Leftist Heap K-ary Heap Quick Links: Practice Problems on Heap Quizzes on Heap Recommended: Learn Data Structure and Algorithms | DSA Tutorial Comment More info Campus Training Program Next Article Introduction to Heap - Data Structure and Algorithm Tutorials course-img 344k+ interested Geeks Data Structures & Algorithms in Python - Self Paced Explore course-img 112k+ interested Geeks Data Structures & Algorithms in JavaScript - Self Paced Course Explore course-img 1k+ interested Geeks Gate CSE 2028 [Semester & Placement Preparation] Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Sure, Manoj! Here's a **concise, MAANG-ready summary** of the **Heap Data Structure** based on the content you shared — optimized for interview prep and system design understanding:

**Heap Data Structure — MAANG Interview Notes**

**🛠 What is a Heap?**

* A **Heap** is a **complete binary tree** satisfying the **heap property**:
  + **Min-Heap:** Parent node ≤ children.
  + **Max-Heap:** Parent node ≥ children.
* The heap is commonly implemented using arrays for efficient indexing.

**🔑 Key Properties**

| **Property** | **Description** |
| --- | --- |
| Complete Binary Tree | All levels filled except possibly last, filled from left to right |
| Heap Property | Parent node compares appropriately to children (min or max) |
| Root | Always the **minimum** (min-heap) or **maximum** (max-heap) element |

**🏗 Types of Heaps**

| **Type** | **Description / Use case** |
| --- | --- |
| **Binary Heap** | Standard min/max heap, supports priority queue efficiently |
| **Binomial Heap** | Supports faster union operations |
| **Fibonacci Heap** | Amortized efficient for decrease-key and merge, used in Dijkstra’s algorithm |
| **Leftist Heap** | Maintains balance with shortest path property |
| **K-ary Heap** | Generalization with k children per node |

**⏱ Time Complexity Summary**

| **Operation** | **Binary Heap (avg)** | **Fibonacci Heap (amortized)** |
| --- | --- | --- |
| Insert | O(log n) | O(1) |
| Extract Min/Max | O(log n) | O(log n) |
| Decrease Key | O(log n) | O(1) |
| Merge (Union) | O(n) | O(1) |
| Build Heap | O(n) | N/A |

**📚 Applications of Heaps**

* Implementing **Priority Queues**
* **Heap Sort** (in-place sorting with O(n log n))
* **Dijkstra’s shortest path** algorithm (with Fibonacci Heap for optimization)
* **Huffman Coding** (optimal prefix codes)
* **Median maintenance** in streaming data
* Efficiently finding **k smallest/largest** elements
* Merging multiple sorted lists/arrays
* Scheduling and resource management in OS and networks

**📖 Common Problems & Concepts**

**Easy**

* Check if an array is a heap
* Kth smallest/largest element
* Heap sort

**Medium**

* Nearly sorted array sorting
* Median in a stream
* Huffman coding tree construction
* Connect ropes with minimum cost

**Hard**

* Merge k sorted arrays/lists
* Skyline problem
* Dijkstra’s shortest path
* Prim’s Minimum Spanning Tree
* Convert BST to Min Heap
* Rearrange characters with constraints

**🛠 Heap Implementations in Languages**

* **C++:** priority\_queue
* **Java:** PriorityQueue
* **Python:** heapq
* **C#:** PriorityQueue
* **JavaScript:** No native heap; often implemented via arrays or libraries

**🔥 Interview Tips**

* Understand how to **build a heap** from an unsorted array (heapify).
* Master **heap insert** and **extract-min/max** operations.
* Practice **heap sort** and **priority queue** implementation.
* Know when to use **Fibonacci Heap** or **Binomial Heap** for optimization.
* Be ready to solve problems involving **k largest/smallest elements**, **merging heaps**, and **graph algorithms** using heaps.

Would you like me to prepare the **Graph Data Structure** summary next, or focus on any particular heap problem for practice?

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... M DSA Course DSA Interview Problems on Graph Practice Graph MCQs on Graph Graph Tutorial Graph Representation Graph Properties Types of Graphs Graph Applications BFS on Graph DFS on Graph Graph VS Tree Transpose Graph Dijkstra's Algorithm Minimum Spanning Tree Prim’s Algorithm Topological Sorting Floyd Warshall Algorithm Strongly Connected Components Advantages & Disadvantages Explore GfG Courses Share Your Experiences DSA Tutorial - Learn Data Structures and Algorithms Basics & Prerequisites Data Structures Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Interview Preparation Practice Problem DSA to DevelopmentCourse Graph Algorithms Last Updated : 21 Jun, 2025 Graph is a non-linear data structure like tree data structure. The limitation of tree is, it can only represent hierarchical data. For situations where nodes or vertices are randomly connected with each other other, we use Graph. Example situations where we use graph data structure are, a social network, a computer network, a network of locations used in GPS and many more examples where different nodes or vertices are connected without any hierarchic or constraint on structure. Basics Graph and its representations BFS and DFS Breadth First Traversal Depth First Traversal Difference between BFS and DFS Rotten Tomatoes Islands in a Graph Flood Fill Check for Bipartite Word Ladder Snakes and Ladder Water Jug problem Pacific Atlantic Water Flow Shortest Path in Binary Matrix Clone a Graph Transitive Closure of a Graph using DFS Cycles Cycle in a Directed Graph Cycle in an undirected graph Cycle in a graph using colors Negative cycle in a Graph | (Bellman Ford) Cycles of length n Clone a Directed Acyclic Graph Disjoint Set Data Structure or Union-Find Algorithm Shortest Path Dijkstra’s shortest path a Bellman–Ford Floyd Warshall Johnson’s algorithm Shortest Path in Directed Acyclic Graph Dial’s Algorithm Multistage Graph (Shortest Path) Shortest path in an unweighted graph Minimum mean weight cycle algorithm 0-1 BFS (Shortest Path Minimum weight cycle D’Esopo-Pape Algorithm Minimum Spanning Tree Prim’s Minimum Spanning Tree (MST) Kruskal’s Minimum Spanning Tree Prim’s vs Kruskal’s algorithm for MST Applications of Minimum Spanning Tree Problem Minimum cost to connect all cities Total number of Spanning Trees in a Graph Minimum Product Spanning Tree Reverse Delete Algorithm for Minimum Spanning Tree Boruvka’s algorithm for Minimum Spanning Tree Topological Sorting Topological Sorting All topological sorts of a Directed Acyclic Graph Kahn’s Algorithm for Topological Sorting Maximum edges that can be added to DAG so that is remains DAG Longest Path in a Directed Acyclic Graph Topological Sort of a graph using departure time of vertex Find Itinerary from a given list of tickets Connectivity in Graph Articulation Points (or Cut Vertices) in a Graph Biconnected Components Bridges in a graph Eulerian path and circuit Fleury’s Algorithm for printing Eulerian Path or Circuit Strongly Connected Components Count all possible walks from a source to a destination with exactly k edges Euler Circuit in a Directed Graph Length of shortest chain to reach the target word Find if an array of strings can be chained to form a circle Tarjan’s Algorithm to find strongly connected Components Paths to travel each nodes using each edge (Seven Bridges of Königsberg) Dynamic Connectivity | Set 1 (Incremental) Maximum Flow in Graph Max Flow Problem Introduction Ford-Fulkerson Algorithm for Maximum Flow Problem Find maximum number of edge disjoint paths between two vertices Find minimum s-t cut in a flow network Maximum Bipartite Matching Channel Assignment Problem Introduction to Push Relabel Algorithm Karger’s Algorithm- Set 1- Introduction and Implementation Dinic’s algorithm for Maximum Flow Some must do Problems Find length of the largest region in Boolean Matrix Count number of trees in a forest A Peterson Graph Problem Clone an Undirected Graph Graph Coloring (Introduction and Applications) Traveling Salesman Problem (TSP) Implementation Vertex Cover Problem | Set 1 (Introduction and Approximate Algorithm) K Centers Problem | Set 1 (Greedy Approximate Algorithm) Erdos Renyl Model (for generating Random Graphs) Chinese Postman or Route Inspection | Set 1 (introduction) Hierholzer’s Algorithm for directed graph Check whether a given graph is Bipartite or not Snake and Ladder Problem Boggle (Find all possible words in a board of characters) Hopcroft Karp Algorithm for Maximum Matching-Introduction Minimum Time to rot all oranges Construct a graph from given degrees of all vertices Determine whether a universal sink exists in a directed graph Number of sink nodes in a graph Two Clique Problem (Check if Graph can be divided in two Cliques) If you are looking for difficulty-wise list of problems, please refer to Graph Data Structure. Some Quizzes Quizzes on Graph Traversal Quizzes on Graph Shortest Path Quizzes on Graph Minimum Spanning Tree Quizzes on Graphs Quick Links : Top 10 Interview Questions on Depth First Search (DFS) Some interesting shortest path questions Practice Problems on Graphs Recommended: Learn Data Structure and Algorithms | DSA Tutorial Comment More info Campus Training Program Next Article Dynamic Programming or DP course-img 344k+ interested Geeks Data Structures & Algorithms in Python - Self Paced Explore course-img 57k+ interested Geeks Data Science Training Program Explore course-img 112k+ interested Geeks Data Structures & Algorithms in JavaScript - Self Paced Course Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Got it! Here's a **MAANG-ready summary** of the **Graph Data Structure** based on the detailed GfG overview you shared — perfect for interviews and system design:

**Graph Data Structure — MAANG Interview Prep Summary**

**🔹 What is a Graph?**

* A **Graph** is a **non-linear data structure** consisting of a set of **nodes (vertices)** connected by **edges**.
* Unlike trees, graphs can model complex relationships without hierarchical constraints.
* Useful for representing networks such as social networks, computer networks, GPS/location maps, and more.

**🔹 Graph Representations**

* **Adjacency Matrix:** 2D matrix storing edges between nodes (good for dense graphs).
* **Adjacency List:** List of neighbors for each vertex (efficient for sparse graphs).
* **Edge List:** List of edges (u, v).

**🔹 Types of Graphs**

| **Type** | **Description** |
| --- | --- |
| **Directed Graph (Digraph)** | Edges have direction (u → v) |
| **Undirected Graph** | Edges have no direction (u ↔ v) |
| **Weighted Graph** | Edges carry weights/costs |
| **Unweighted Graph** | Edges have no weights |
| **Cyclic / Acyclic** | Contains or does not contain cycles |
| **Connected / Disconnected** | Every vertex reachable from every other / not |

**🔹 Core Graph Algorithms**

**Traversals**

* **BFS (Breadth First Search):** Level-order traversal, uses queue.
* **DFS (Depth First Search):** Recursive or stack-based traversal.

**Shortest Path**

* **Dijkstra’s Algorithm:** Shortest path in weighted graphs without negative edges.
* **Bellman-Ford Algorithm:** Handles negative weights, detects negative cycles.
* **Floyd-Warshall:** All-pairs shortest path.
* **Johnson’s Algorithm:** Efficient for sparse graphs with negative weights.

**Minimum Spanning Tree (MST)**

* **Prim’s Algorithm**
* **Kruskal’s Algorithm**
* **Boruvka’s Algorithm**
* Applications in network design, clustering, etc.

**Topological Sorting**

* For Directed Acyclic Graphs (DAG).
* Kahn’s Algorithm (BFS-based).
* DFS based approach.

**🔹 Advanced Concepts**

* **Strongly Connected Components (SCC):** Tarjan’s algorithm.
* **Articulation Points & Bridges:** Identify critical nodes/edges.
* **Cycle Detection:** In directed/undirected graphs.
* **Eulerian Path and Circuit:** Traversing every edge exactly once.
* **Graph Coloring:** Assign colors with constraints.
* **Maximum Flow:** Ford-Fulkerson, Dinic’s Algorithm, Push-Relabel.

**🔹 Common Graph Problems**

* Clone Graph
* Detect Cycle in Directed and Undirected Graphs
* Check Bipartite Graph
* Word Ladder (Shortest transformation sequence)
* Snake and Ladder Problem
* Islands in a Matrix (Connected components)
* Traveling Salesman Problem (TSP)
* Maximum Bipartite Matching (Hopcroft-Karp)
* Minimum Time to Rot All Oranges (BFS variant)

**🔹 Applications of Graphs**

* Social network connections and recommendations
* GPS and routing algorithms
* Network routing and flow optimization
* Dependency resolution and scheduling
* Compiler design (topological sort)
* Biology (gene network analysis)

**🔹 Interview Tips**

* Understand **graph representations** and choose the right one for your problem.
* Master **BFS and DFS** traversal and their use cases.
* Practice **shortest path** and **MST** problems thoroughly.
* Know how to detect **cycles** and handle **directed/undirected** cases.
* Be comfortable with advanced graph algorithms like **Tarjan’s SCC**, **Ford-Fulkerson** for max flow.
* Solve problems on graph connectivity, path finding, and graph transformations.

If you'd like, I can also help you with:

* Code implementations (C++, Python, Java)
* Detailed explanations of any specific graph algorithm
* Practice problems sorted by difficulty
* Visual explanations or pseudo-code for tricky concepts

Just let me know!

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... DSA Course DSA Algorithms Analysis of Algorithms Sorting Searching Greedy Recursion Backtracking Dynamic Programming Divide and Conquer Geometric Algorithms Mathematical Algorithms Pattern Searching Bitwise Algorithms Branch & Bound Randomized Algorithms Explore GfG Courses Share Your Experiences DSA Tutorial - Learn Data Structures and Algorithms Basics & Prerequisites Data Structures Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Interview Preparation Practice Problem DSA to DevelopmentCourse Greedy Algorithms Last Updated : 07 Apr, 2025 Greedy algorithms are a class of algorithms that make locally optimal choices at each step with the hope of finding a global optimum solution. At every step of the algorithm, we make a choice that looks the best at the moment. To make the choice, we sometimes sort the array so that we can always get the next optimal choice quickly. We sometimes also use a priority queue to get the next optimal item. After making a choice, we check for constraints (if there are any) and keep picking until we find the solution. Greedy algorithms do not always give the best solution. For example, in coin change and 0/1 knapsack problems, we get the best solution using Dynamic Programming. Examples of popular algorithms where Greedy gives the best solution are Fractional Knapsack, Dijkstra's algorithm, Kruskal's algorithm, Huffman coding and Prim's Algorithm Basics of Greedy Algorithm Introduction to Greedy Algorithm Greedy Algorithms General Structure Easy Problems on Greedy Algorithm Fractional Knapsack Min Cost to Make Array Size 1 Min Rotations for Circular Lock Max Composite Numbers to Make n Smallest Subset Greater Sum Assign Cookies Buy Maximum Stocks Max Consecutive Diff Sum Min and Max Costs to buy all Min Notes with Given Sum Max Equal Sum of Three Stacks Medium Problems on Greedy Algorithm Activity Selection Problem Jump Game Job Sequencing Problem Egyptian Fraction Merge Overlapping Intervals Min Fibonacci Terms with Sum K Minimum Platforms Min Cost to Connect n ropes Max trains Partition 1 to n into two min diff groups Paper cut into min squares Min diff groups of size two Max Satisfied Customers Min initial vertices to traverse matrix with constraints Largest palindromic number by permuting digits Smallest with n digits and digits sum Lexicographically largest subsequence with every char at least k times Hard Problems on Greedy Algorithm Minimize the Max Height Diff Making max equal with k updates Minimize cash flow among friends Minimum Cost to cut a board into squares Minimum cost to process m tasks where switching costs Minimum time to finish all jobs with given constraints Minimize the maximum difference between the heights of towers Minimum edges to reverse to make path from a source to a destination Find the Largest Cube formed by Deleting minimum Digits from a number Rearrange characters in a string such that no two adjacent are same Rearrange a string so that all same characters become d distance away Standard Greedy Algorithms Activity Selection Problem Job Sequencing Problem Huffman Coding Huffman Decoding Water Connection Problem Minimum Swaps for Bracket Balancing Egyptian Fraction Policemen catch thieves Fitting Shelves Problem Assign Mice to Holes Greedy Problems on Array Minimum product subset of an array Maximize array sum after K negations using Sorting Minimum sum of product of two arrays Minimum sum of absolute difference of pairs of two arrays Minimum increment/decrement to make array non-Increasing Sorting array with reverse around middle Sum of Areas of Rectangles possible for an array Largest lexicographic array with at-most K consecutive swaps Partition into two subarrays of lengths k and (N – k) such that the difference of sums is maximum Greedy Problems in Operating System First Fit algorithm in Memory Management Best Fit algorithm in Memory Management Worst Fit algorithm in Memory Management Shortest Job First Scheduling Job Scheduling with two jobs allowed at a time Program for Optimal Page Replacement Algorithm Greedy Problems on Graph Kruskal’s Minimum Spanning Tree Prim’s Minimum Spanning Tree Boruvka’s Minimum Spanning Tree Dijkastra’s Shortest Path Algorithm Dial’s Algorithm Minimum cost to connect all cities Max Flow Problem Introduction Number of single cycle components in an undirected graph Approximate Greedy Algorithm for NP Complete Set cover problem Bin Packing Problem Graph Coloring K-centers problem Shortest superstring problem Approximate solution for Travelling Salesman Problem using MST Greedy for Special cases of DP Fractional Knapsack Problem Minimum number of coins required Quick Links Learn Data Structure and Algorithms | DSA Tutorial Top 20 Greedy Algorithms Interview Questions ‘Practice Problems’ on Greedy Algorithms ‘Quiz’ on Greedy Algorithms Comment More info Campus Training Program Next Article Graph Algorithms course-img 1418k+ interested Geeks Data Structures and Algorithms - Self Paced [Online Course] Explore course-img 112k+ interested Geeks Data Structures & Algorithms in JavaScript - Self Paced Course Explore course-img 344k+ interested Geeks Data Structures & Algorithms in Python - Self Paced Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Here’s a concise, interview-ready summary on **Greedy Algorithms** based on your detailed GfG content — great for quick revision or prep for MAANG-style interviews:

**Greedy Algorithms — MAANG Interview Prep Summary**

**🔹 What is a Greedy Algorithm?**

* Greedy algorithms make **locally optimal choices** at each step, hoping to reach a **global optimum**.
* They pick the best available option at each step without reconsidering past choices.
* Often require sorting or priority queues to efficiently pick the next optimal item.
* **Do not always guarantee the best solution** (example: 0/1 Knapsack), but work well in many cases.

**🔹 When to Use Greedy?**

* When the problem has the **Greedy Choice Property**: a global optimum can be arrived by choosing local optimums.
* When the problem exhibits **Optimal Substructure**: an optimal solution to the problem contains optimal solutions to subproblems.

**🔹 Common Examples Where Greedy Works Perfectly**

* **Fractional Knapsack**
* **Dijkstra’s Shortest Path Algorithm**
* **Kruskal’s and Prim’s Minimum Spanning Tree (MST) Algorithms**
* **Huffman Coding**

**🔹 General Structure of Greedy Algorithms**

1. **Sort** or preprocess input data if needed.
2. Initialize your solution set.
3. Iterate over elements, pick the current best candidate (local optimum).
4. Check feasibility or constraints.
5. Add the candidate to solution if valid.
6. Repeat until solution is complete or no more candidates.

**🔹 Popular Greedy Problems (by Difficulty)**

**Easy**

* Fractional Knapsack
* Min Rotations for Circular Lock
* Assign Cookies
* Min Notes with Given Sum

**Medium**

* Activity Selection Problem
* Job Sequencing Problem
* Merge Overlapping Intervals
* Minimum Platforms (Train platforms)
* Min Cost to Connect N Ropes

**Hard**

* Minimum Cost to Cut a Board into Squares
* Minimum Time to Finish All Jobs with Constraints
* Rearrange Characters so No Two Adjacent are Same
* Minimize Cash Flow Among Friends
* Minimum Edges to Reverse to Make Path

**🔹 Greedy Algorithms in Graphs**

* **Kruskal’s MST**
* **Prim’s MST**
* **Boruvka’s MST**
* **Dijkstra’s Shortest Path**
* Approximate solutions to NP-complete problems like Set Cover, Graph Coloring, Traveling Salesman (via MST).

**🔹 Tips for Interview Preparation**

* Understand **why greedy works** for a problem — prove greedy choice property and optimal substructure.
* Practice classical greedy problems to build intuition.
* Learn how greedy algorithms differ from dynamic programming (when greedy fails).
* Be comfortable explaining time and space complexity.
* Implement algorithms both using sorting and priority queues.
* Solve problems involving scheduling, intervals, and resource allocation.

If you want, I can help with:

* Code walkthroughs of classic greedy algorithms (in C++, Python, or Java).
* Explanation of where greedy fails and DP is needed.
* Practice problems with solutions and explanations.
* Comparison of greedy vs DP for common problems.

Just ask!

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... 17 M DSA Course DSA Tutorial Data Structures Algorithms Array Strings Linked List Stack Queue Tree Graph Searching Sorting Recursion Dynamic Programming Binary Tree Binary Search Tree Heap Hashing Divide & Conquer Mathematical Geometric Bitwise Greedy Backtracking Branch and Bound Matrix Pattern Searching Randomized ▲ Explore GfG Courses Share Your Experiences DSA Tutorial - Learn Data Structures and Algorithms Basics & Prerequisites Data Structures Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Interview Preparation Practice Problem DSA to DevelopmentCourse Dynamic Programming or DP Last Updated : 18 Mar, 2025 Dynamic Programming is an algorithmic technique with the following properties. It is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for the same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems so that we do not have to re-compute them when needed later. This simple optimization typically reduces time complexities from exponential to polynomial. Some popular problems solved using Dynamic Programming are Fibonacci Numbers, Diff Utility (Longest Common Subsequence), Bellman–Ford Shortest Path, Floyd Warshall, Edit Distance and Matrix Chain Multiplication. Dynamic-Programming-or-DP-1.webpDynamic-Programming-or-DP-1.webp Basic of DP Introduction to DP Tabulation vs Memoization Steps to solve a DP Problem Basic Problems Fibonacci numbers Tribonacci Numbers Lucas Numbers Climbing Stairs Climbing Stairs with 3 Moves Weighted Climbing Stairs Maximum Segments nth Catalan Number Count Unique BSTs Count Valid Parenthesis Ways to Triangulate a Polygon Min Sum in a Triangle Minimum Perfect Squares Ways to Partition a Set Binomial Coefficient Pascal's Triangle Nth Row of Pascal Triangle Min Sum in a Triangle Easy Problems House Robber Min Cost Path Decode Ways Subset Sum Problem Coin change problem - Count Ways Coin Change – Minimum Coins to Make Sum Painting Fence Algorithm Cutting a Rod Jump Game Longest Common Substring Count all paths in a Grid Paths in a Grid with Obstacles Permutations with K Inversions Max A's using Special Keyboard Medium Problems Water Overflow Longest Common Subsequence Longest Increasing Subsequence Edit Distance Largest Divisible Subset Weighted Job Schedulling 0-1 Knapsack Problem Printing Items in 0/1 Knapsack Unbounded Knapsack Word Break Problem Tile Stacking Problem Box-Stacking Problem Partition Problem Longest Palindromic Subsequence Longest Common Increasing Subsequence (LCS + LIS) All distinct subset (or subsequence) sums Count Derangements Minimum insertions for palindrome Wildcard Pattern Matching Regular Expression Matching Arrange Balls with adjacent of different types Longest Subsequence with 1 adjacent difference Maximum size square sub-matrix with all 1s Bellman–Ford Algorithm Floyd Warshall Algorithm Maximum Tip Calculator Hard Problems Largest X Bordered Square Egg Dropping Problem Palindrome Partitioning Palindromic Substring Count Word Wrap Problem Optimal Strategy for a Game The painter’s partition problem Program for Bridge and Torch problem Matrix Chain Multiplication Printing Matrix Chain Multiplication Maximum sum rectangle Stock Buy and Sell - At-Most k Times Stock Buy and Sell - At Most 2 Times Min cost to sort strings using Reversals Count of AP Subsequences DP on Trees Max Height of Tree when any Node can be Root Longest repeating and non-overlapping substring Palindrome Substrings Count DP Problems Sorted by Topic / Dimensions / Standard Problems DP Standard Problems and Variations. DP Problems Dimension Wise (1D, 2D and 3D) DP Problems Topic Wise Advanced Concepts in Dynamic Programming (DP) Bitmasking and Dynamic Programming | Set 1 Bitmasking and Dynamic Programming | Set-2 (TSP) Digit DP | Introduction Sum over Subsets | Dynamic Programming Quick Links: Learn Data Structure and Algorithms | DSA Tutorial Top 20 Dynamic Programming Interview Questions ‘Practice Problems’ on Dynamic Programming ‘Quiz’ on Dynamic Programming Comment More info Campus Training Program Next Article Bitwise Algorithms course-img 977k+ interested Geeks Complete Interview Preparation Explore course-img 473k+ interested Geeks Complete Machine Learning & Data Science Program Explore course-img 200k+ interested Geeks C Programming Course Online - Learn C with Data Structures Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Here’s a concise and interview-focused summary on **Dynamic Programming (DP)** based on the content you shared, perfect for quick learning or interview prep:

**Dynamic Programming (DP) — Interview Prep Summary**

**🔹 What is Dynamic Programming?**

* **DP** is an optimization technique to improve recursive solutions with overlapping subproblems.
* It stores results of subproblems (memoization or tabulation) to **avoid recomputation**.
* Converts exponential time recursive solutions to **polynomial time**.

**🔹 When to Use DP?**

* The problem exhibits:
  + **Overlapping Subproblems**: Same subproblems are solved multiple times.
  + **Optimal Substructure**: Optimal solution can be constructed from optimal solutions of subproblems.

**🔹 Basic Approaches**

* **Memoization (Top-Down)**: Recursion + caching intermediate results.
* **Tabulation (Bottom-Up)**: Iterative approach building up the solution from smallest subproblems.

**🔹 Steps to Solve a DP Problem**

1. **Define subproblems:** Identify parameters representing the state.
2. **Guess:** Consider possible decisions/choices.
3. **Recurrence relation:** Express solution in terms of smaller subproblems.
4. **Base cases:** Define simplest subproblem solutions.
5. **Compute & store:** Use memoization or tabulation to save intermediate results.

**🔹 Popular Classic Problems (by Difficulty)**

**Easy**

* Fibonacci Numbers
* Climbing Stairs
* Count Unique BSTs
* Minimum Perfect Squares
* Coin Change (Count ways, Minimum coins)
* House Robber
* Subset Sum Problem

**Medium**

* Longest Common Subsequence (LCS)
* Edit Distance
* 0-1 Knapsack Problem
* Unbounded Knapsack
* Word Break
* Longest Increasing Subsequence
* Partition Problem
* Weighted Job Scheduling
* Bellman-Ford Algorithm
* Floyd Warshall Algorithm

**Hard**

* Egg Dropping Problem
* Matrix Chain Multiplication
* Palindrome Partitioning
* Word Wrap Problem
* Painter’s Partition Problem
* Stock Buy and Sell (At most k times)
* Count of Arithmetic Progression Subsequences
* DP on Trees
* Bitmasking + DP (e.g., Traveling Salesman Problem)

**🔹 Important Concepts & Variants**

* **1D, 2D, 3D DP:** Problems can have states defined by 1 or more parameters.
* **Bitmask DP:** For subset-based problems, often in combinatorial optimization.
* **Digit DP:** Counting numbers with constraints digit-wise.
* **DP on Trees:** Problems where subproblems are defined on tree nodes.
* **Advanced:** Sum over subsets, DP with optimization tricks.

**🔹 Tips for Interview**

* Practice identifying **states** and **transition formulas**.
* Understand difference between memoization and tabulation; when to use which.
* Solve standard problems thoroughly to build intuition.
* Be ready to explain time and space complexity.
* Optimize space if possible (rolling arrays).
* Learn bitmask and digit DP for harder combinatorial problems.

If you want, I can provide:

* Step-by-step solution and code for classic DP problems (Fibonacci, Knapsack, LCS, etc.).
* Explanation and walkthrough of advanced DP concepts like bitmask DP.
* Practice problems with detailed explanations.
* Comparisons of DP vs Greedy vs Recursion.

Just let me know!

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... M DSA Course DSA Practice Bitwise Algorithms MCQs on Bitwise Algorithms Tutorial on Biwise Algorithms Binary Representation Bitwise Operators Bit Swapping Bit Manipulation Count Set bits Setting a Bit Clear a Bit Toggling a Bit Left & Right Shift Gray Code Checking Power of 2 Important Tactics Bit Manipulation for CP Fast Exponentiation Explore GfG Courses Share Your Experiences DSA Tutorial - Learn Data Structures and Algorithms Basics & Prerequisites Data Structures Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Interview Preparation Practice Problem DSA to DevelopmentCourse Bitwise Algorithms Last Updated : 07 Apr, 2025 Bitwise algorithms in Data Structures and Algorithms (DSA) involve manipulating individual bits of binary representations of numbers to perform operations efficiently. These algorithms utilize bitwise operators like AND, OR, XOR, NOT, Left Shift, and Right Shift. Basics Introduction to Bitwise Algorithms Bitwise Operators in C/C++ Bitwise Operators in Java Python Bitwise Operators JavaScript Bitwise Operators All about Bit Manipulation Little and Big Endian Mystery Bit Manipulation Tips and Tricks Bits manipulation (Important tactics) Bitwise Hacks for Competitive Programming Easy Problems on Bit Algorithms Binary Representation Turn off the rightmost set bit Check if K-th Bit Set Set the K-th Bit Modulus division by 2's Power Odd Occurring Number Power of two The only set bit Add Bit Strings Check for Integer Overflow XOR without using XOR Check for Equal Check for opposite signs Swap Two Numbers Russian Peasant Medium Problems on Bit Algorithms Most Significant Set Bit Rightmost Set Bit Count Set Bits Swap Bits Rotate Bits Smallest of Three Minimum without branching Smallest power of 2 greater than or equal to n Program to find parity Check if binary is palindrome Generate n-bit Gray Codes Check for Sparse Euclid when % and / are costly Square without using \*, / and pow() Cyclic Redundancy Check and Modulo-2 Division Set Bits in a Range Check for Bleak Gray to Binary and Vice Versa Hard Problems on Bit Algorithms Next higher with same set bits Karatsuba Algorithm for fast Multiplication Max Subarray XOR Longest Sequence of 1’s in Binary with One Flip Closest Smaller and greater with same set bits Bitmasking and Dynamic Programming Compute the Parity XOR Encryption by Shifting Plaintext Count pairs with at least one digit common Floating to Binary Booth’s Multiplication Algorithm Pairs with Pandigital Concatenation n-th number whose binary is a palindrome Two non-repeating in an array of repeating Quick Links : DSA Tutorial ‘Practice Problems’ on Bit Magic ‘Quiz’ on Bit Magic ‘Videos’ on Bit Magic What are Bitwise Algorithms? Bitwise algorithms are algorithms that operate on individual bits of data rather than on larger data types like integers or floating-point numbers. These algorithms manipulate bits directly, typically using bitwise operators such as AND, OR, XOR, shift left, shift right, and complement. Common Bitwise Algorithms and Operations Here are some common bitwise algorithms and operations: Bitwise AND (&): Takes two numbers as input and performs a bitwise AND operation on their corresponding bits. It returns 1 only if both bits are 1; otherwise, it returns 0. Bitwise OR (|): Performs a bitwise OR operation on the corresponding bits of two numbers. It returns 1 if at least one of the bits is 1. Bitwise XOR (^): Performs a bitwise exclusive OR operation on the corresponding bits of two numbers. It returns 1 if the bits are different and 0 if they are the same. Bitwise NOT (~): Performs a bitwise NOT operation, which flips each bit of the input (1 becomes 0 and 0 becomes 1). Left Shift (<<) and Right Shift (>>): These operators shift the bits of a number to the left or right by a specified number of positions. Left shifting is equivalent to multiplying the number by 2, while right shifting is equivalent to dividing by 2. Applications of Bitwise Algorithms Bit manipulation (setting, clearing, toggling bits): Bitwise operators are often used to manipulate individual bits of numbers. This includes tasks such as setting bits (using OR), clearing bits (using AND with the complement), toggling bits (using XOR with 1), and checking the value of a specific bit. Efficient storage of data: Bitwise algorithms play a crucial role in data compression techniques like Huffman coding. They can efficiently represent and process compressed data by manipulating bits directly. Cryptography: Many cryptographic algorithms, such as AES (Advanced Encryption Standard), DES (Data Encryption Standard), and SHA (Secure Hash Algorithm), utilize bitwise operations for encryption, decryption, and hashing. Bitwise XOR, in particular, is commonly used in encryption algorithms for its simplicity and effectiveness. Networking and Protocol Handling: Bitwise algorithms are used in networking protocols for tasks like IP address manipulation, subnet masking, and packet parsing. For example, bitwise AND is used in subnet masking to determine the network address from an IP address and subnet mask. Low-Level System Programming: Bitwise operations are essential in low-level system programming for tasks such as device control, memory management, and bit-level I/O operations. They are used to manipulate hardware registers, set/clear flags, and optimize code for performance. Error Detection and Correction: Bitwise algorithms are employed in error detection and correction techniques, such as CRC (Cyclic Redundancy Check) and Hamming codes. These algorithms use bitwise XOR and other operations to detect and correct errors in transmitted data. Quick Links : DSA Tutorial ‘Practice Problems’ on Bit Magic ‘Quiz’ on Bit Magic ‘Videos’ on Bit Magic Comment More info Campus Training Program Next Article Segment Tree course-img 1417k+ interested Geeks Data Structures and Algorithms - Self Paced [Online Course] Explore course-img 112k+ interested Geeks Data Structures & Algorithms in JavaScript - Self Paced Course Explore course-img 344k+ interested Geeks Data Structures & Algorithms in Python - Self Paced Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Here’s a clear and concise overview of **Bitwise Algorithms** from the content you shared, ideal for quick study or interview prep:

**Bitwise Algorithms — Overview**

**🔹 What Are Bitwise Algorithms?**

* Algorithms that **manipulate individual bits** of numbers directly.
* Use **bitwise operators** like AND (&), OR (|), XOR (^), NOT (~), Left Shift (<<), and Right Shift (>>).
* Operate at a **binary level** for efficient computation.

**🔹 Common Bitwise Operators & Their Meaning**

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| & | Bitwise AND | 5 & 3 = 1 (0101 & 0011) |
| ` | ` | Bitwise OR |
| ^ | Bitwise XOR (exclusive OR) | 5 ^ 3 = 6 (0101 ^ 0011) |
| ~ | Bitwise NOT (complement) | ~5 = -6 (in two’s complement) |
| << | Left Shift (multiply by 2^n) | 5 << 1 = 10 |
| >> | Right Shift (divide by 2^n) | 5 >> 1 = 2 |

**🔹 Common Bitwise Operations & Problems**

* **Set, clear, toggle bits:**  
  Set: num |= (1 << k)  
  Clear: num &= ~(1 << k)  
  Toggle: num ^= (1 << k)
* **Check if k-th bit is set:**  
  (num & (1 << k)) != 0
* **Count number of set bits (1s) in a number.**
* **Turn off rightmost set bit:**  
  num = num & (num - 1)
* **Check if a number is power of two:**  
  num > 0 && (num & (num - 1)) == 0
* **Swap two numbers without temporary variable:**  
  Using XOR swap.

**🔹 Applications of Bitwise Algorithms**

* **Efficient data storage & compression:** Manipulate bits directly to save space.
* **Cryptography:** XOR operations for encryption/decryption.
* **Networking:** IP address manipulation, subnet masks.
* **Low-level system programming:** Hardware register control, flag settings.
* **Error detection & correction:** Cyclic Redundancy Check (CRC), Hamming codes.

**🔹 Examples of Bitwise Algorithm Problems**

* Count set bits in an integer.
* Generate Gray codes.
* Find the parity of a number.
* Find the next higher number with the same number of set bits.
* Max subarray XOR.
* Detect whether two integers have opposite signs.
* Check if a binary string is a palindrome.
* Compute XOR encryption.
* Bitmasking combined with dynamic programming for subset problems.

**🔹 Tips for Interviews**

* Understand how each bitwise operator works on binary.
* Practice common tricks like clearing the rightmost set bit, counting bits, and swapping numbers.
* Be able to use bitmasking for subset and combinatorial problems.
* Know how to optimize algorithms using bitwise operations where applicable.
* Solve classic bitwise problems to gain intuition.

If you want, I can provide code examples, explanations for specific bitwise tricks, or help with common bitwise interview problems. Just ask!

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... M DSA Course DSA Practice Backtracking Interview Problems on Backtracking MCQs on Backtracking Tutorial on Backtracking Backtracking vs Recursion Backtracking vs Branch & Bound Print Permutations Subset Sum Problem N-Queen Problem Knight's Tour Sudoku Solver Rat in Maze Hamiltonian Cycle Graph Coloring ▲ Explore GfG Courses Share Your Experiences Backtracking Algorithm Introduction to Backtracking Difference between Backtracking and Branch-N-Bound technique What is the difference between Backtracking and Recursion? Standard problems on backtracking Easy Problems on Backtracking Medium prblems on Backtracking Hard problems on Backtracking Top 20 Backtracking Algorithm Interview Questions DSA to DevelopmentCourse Backtracking Algorithm Last Updated : 01 Dec, 2024 Backtracking algorithms are like problem-solving strategies that help explore different options to find the best solution. They work by trying out different paths and if one doesn't work, they backtrack and try another until they find the right one. It's like solving a puzzle by testing different pieces until they fit together perfectly. Backtracking Basic of Backtracking Algorithm: Introduction to Backtracking Backtracking vs Recursion Backtracking vs Branch and Bound Standard Problems Permutations of a String The Knight’s tour problem Rat in a Maze N Queen Problem | Backtracking-3 Subset Sum problem m Coloring Problem Hamiltonian Cycle Sudoku | Backtracking-7 Magnet Puzzle Remove Invalid Parentheses A backtracking approach to generate n bit Gray Codes Easy Problem: Backtracking to find all subsets Check if a given string is sum-string Count all possible paths between two vertices Find all distinct subsets of a given set Find if there is a path of more than k length from a source Print all paths from a given source to a destination Print all possible strings that can be made by placing spaces Medium Problems: Tug of War 8 queen problem Combinational Sum Warnsdorff’s algorithm for Knight’s tour problem Find paths from corner cell to middle cell in maze Find Maximum number possible by doing at-most K swaps Rat in a Maze with multiple steps or jump allowed N Queen in O(n) space Hard Problems: Power Set in Lexicographic order Word Break Problem using Backtracking Partition of a set into K subsets with equal sum Longest Possible Route in a Matrix with Hurdles Find shortest safe route in a path with landmines Print all palindromic partitions of a string Printing all solutions in N-Queen Problem Print all longest common sub-sequences in lexicographical order Quick Links : Learn Data Structure and Algorithms | DSA Tutorial Top 20 Backtracking Algorithm Interview Questions ‘Practice Problems’ on Backtracking ‘Quiz’ on Backtracking ‘Videos’ on Backtracking What is Backtracking Algorithm? Backtracking is a problem-solving algorithmic technique that involves finding a solution incrementally by trying different options and undoing them if they lead to a dead end. It is commonly used in situations where you need to explore multiple possibilities to solve a problem, like searching for a path in a maze or solving puzzles like Sudoku. When a dead end is reached, the algorithm backtracks to the previous decision point and explores a different path until a solution is found or all possibilities have been exhausted. How Does a Backtracking Algorithm Work? A backtracking algorithm works by recursively exploring all possible solutions to a problem. It starts by choosing an initial solution, and then it explores all possible extensions of that solution. If an extension leads to a solution, the algorithm returns that solution. If an extension does not lead to a solution, the algorithm backtracks to the previous solution and tries a different extension. The following is a general outline of how a backtracking algorithm works: Choose an initial solution. Explore all possible extensions of the current solution. If an extension leads to a solution, return that solution. If an extension does not lead to a solution, backtrack to the previous solution and try a different extension. Repeat steps 2-4 until all possible solutions have been explored. Example of Backtracking Algorithm Example: Finding the shortest path through a maze Input: A maze represented as a 2D array, where 0 represents an open space and 1 represents a wall. Algorithm: Start at the starting point. For each of the four possible directions (up, down, left, right), try moving in that direction. If moving in that direction leads to the ending point, return the path taken. If moving in that direction does not lead to the ending point, backtrack to the previous position and try a different direction. Repeat steps 2-4 until the ending point is reached or all possible paths have been explored. When to Use a Backtracking Algorithm? Backtracking algorithms are best used to solve problems that have the following characteristics: There are multiple possible solutions to the problem. The problem can be broken down into smaller subproblems. The subproblems can be solved independently. Applications of Backtracking Algorithm Backtracking algorithms are used in a wide variety of applications, including: Solving puzzles (e.g., Sudoku, crossword puzzles) Finding the shortest path through a maze Scheduling problems Resource allocation problems Network optimization problems Combinatorial problems, such as generating permutations, combinations, or subsets. Comment More info Campus Training Program Next Article Introduction to Backtracking course-img 112k+ interested Geeks Data Structures & Algorithms in JavaScript - Self Paced Course Explore course-img 1418k+ interested Geeks Data Structures and Algorithms - Self Paced [Online Course] Explore course-img 344k+ interested Geeks Data Structures & Algorithms in Python - Self Paced Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Here’s a concise summary of **Backtracking Algorithm** from the content you shared, ideal for quick understanding or interview prep:

**Backtracking Algorithm — Overview**

**🔹 What is Backtracking?**

* **Backtracking** is a problem-solving technique that incrementally builds candidates to the solution and abandons a candidate (“backtracks”) as soon as it determines that the candidate cannot lead to a valid solution.
* Think of it like solving a puzzle by trying pieces and undoing moves when a dead end is reached, then trying another path.

**🔹 How Does Backtracking Work?**

1. Start with an initial partial solution.
2. Try to extend the current solution by exploring all possible choices.
3. If an extension leads to a solution, return it.
4. If an extension leads to a dead end (no solution), backtrack (undo the last step) and try other alternatives.
5. Repeat until all solutions are found or all possibilities exhausted.

**🔹 When to Use Backtracking?**

* When there are **multiple possible solutions**.
* When the problem can be divided into **smaller independent subproblems**.
* When you need to explore **all or some of the combinations** (e.g., permutations, subsets).
* Useful in **constraint satisfaction problems**.

**🔹 Applications of Backtracking**

* Solving puzzles: Sudoku, crossword, N-Queens.
* Finding paths through mazes.
* Scheduling and resource allocation.
* Combinatorial problems: generating permutations, subsets, combinations.
* Graph problems like Hamiltonian cycle, graph coloring.

**🔹 Standard Backtracking Problems**

* N-Queens Problem
* Rat in a Maze
* Knight’s Tour Problem
* Sudoku Solver
* Subset Sum Problem
* Graph Coloring
* Hamiltonian Cycle
* Permutations of a string

**🔹 Difference from Recursion & Branch & Bound**

* **Backtracking** is a refined form of recursion that tries all possibilities but prunes invalid paths early.
* **Branch & Bound** is similar but uses bounds to prune branches that cannot yield better solutions than already found.

If you'd like, I can provide sample code snippets, explain any particular problem, or help you practice backtracking questions!

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... M DSA Course DSA Practice Divide and Conquer MCQs on Divide and Conquer Tutorial on Divide & Conquer Binary Search Merge Sort Quick Sort Calculate Power Strassen's Matrix Multiplication Karatsuba Algorithm Divide and Conquer Optimization Closest Pair of Points Explore GfG Courses Share Your Experiences Divide and Conquer Algorithm Introduction to Divide and Conquer Algorithm Dynamic Programming vs Divide-and-Conquer Decrease and Conquer Advanced master theorem for divide and conquer recurrences Some standard Divide and Conquer Algorithms Binary Search based problems Some practice problems on Divide and Conquer algorithm DSA to DevelopmentCourse Divide and Conquer Algorithm Last Updated : 15 Nov, 2024 Divide and Conquer algorithm is a problem-solving strategy that involves. Divide : Break the given problem into smaller non-overlapping problems. Conquer : Solve Smaller Problems Combine : Use the Solutions of Smaller Problems to find the overall result. Examples of Divide and Conquer are Merge Sort, Quick Sort, Binary Search and Closest Pair of Points. Divide-and-Conquer-banner Basics of Divide and Conquer Algorithm: Introduction to Divide and Conquer Dynamic Programming vs Divide-and-Conquer Decrease and Conquer Advanced master theorem for divide and conquer recurrences Standard Algorithms on Divide and Conquer Algorithm: Binary Search Merge Sort Quick Sort Calculate pow(x, n) Karatsuba algorithm for fast multiplication Strassen’s Matrix Multiplication Convex Hull (Simple Divide and Conquer Algorithm) Quickhull Algorithm for Convex Hull Practice problems on Divide and Conquer Algorithm: Square root of an integer Maximum and minimum of an array using minimum number of comparisons Find frequency of each element in a limited range array in less than O(n) time Tiling Problem Count Inversions The Skyline Problem Search in a Row-wise and Column-wise Sorted 2D Array Allocate minimum number of pages Modular Exponentiation (Power in Modular Arithmetic) Problems Based on Standard Divide and Conquer Algorithms Binary Search Based Problems Merge Sort Based Problems Quick Sort Based Problems Quick Links : Learn Data Structure and Algorithms | DSA Tutorial ‘Practice Problems’ on Divide and Conquer ‘Quizzes’ on Divide and Conquer Comment More info Campus Training Program Next Article Introduction to Divide and Conquer Algorithm course-img 112k+ interested Geeks Data Structures & Algorithms in JavaScript - Self Paced Course Explore course-img 1418k+ interested Geeks Data Structures and Algorithms - Self Paced [Online Course] Explore course-img 344k+ interested Geeks Data Structures & Algorithms in Python - Self Paced Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Here's a clear summary of the **Divide and Conquer Algorithm** concept from the content you provided:

**Divide and Conquer Algorithm — Overview**

**What is Divide and Conquer?**

Divide and Conquer is a powerful problem-solving paradigm that breaks down a problem into smaller subproblems, solves each subproblem independently, and then combines their solutions to solve the original problem.

**Steps:**

1. **Divide**: Break the problem into smaller, non-overlapping subproblems.
2. **Conquer**: Solve each smaller problem recursively (or directly if small enough).
3. **Combine**: Merge the results of the subproblems to get the final solution.

**Key Examples of Divide and Conquer Algorithms**

* **Binary Search**
* **Merge Sort**
* **Quick Sort**
* **Calculate Power (Exponentiation)**
* **Karatsuba Multiplication (fast multiplication of large numbers)**
* **Strassen’s Matrix Multiplication**
* **Closest Pair of Points Problem**

**Divide and Conquer vs Dynamic Programming**

* **Divide and Conquer** solves problems by dividing and solving subproblems independently without overlap.
* **Dynamic Programming** is used when subproblems overlap and results are reused (memoization/tabulation).

**Advanced Topics**

* Master Theorem for analyzing time complexity of Divide and Conquer recurrences.
* Decrease and Conquer — a variation where the problem size is reduced by a constant amount instead of half.

**Practice Problems on Divide and Conquer**

* Square root of an integer
* Finding maximum and minimum in an array with minimum comparisons
* Counting inversions in an array
* Tiling problem
* The Skyline problem
* Searching in a row-wise and column-wise sorted 2D array
* Modular exponentiation

If you'd like, I can explain any of these algorithms in detail or provide code examples for better understanding!

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... M DSA Course DSA Branch and Bound Tutorial Backtracking Vs Branch-N-Bound 0/1 Knapsack 8 Puzzle Problem Job Assignment Problem N-Queen Problem Travelling Salesman Problem Explore GfG Courses Share Your Experiences Algorithms Tutorial What is an Algorithm | Introduction to Algorithms Definition, Types, Complexity and Examples of Algorithm Algorithms Design Techniques Why is Analysis of Algorithm important? Analysis of Algorithms Types of Algorithms Sorting Algorithms Searching Algorithms Greedy Algorithms Dynamic Programming or DP What is Pattern Searching ? Backtracking Algorithm Divide and Conquer Algorithm Mathematical Algorithms Geometric Algorithms Bitwise Algorithms Graph Algorithms Randomized Algorithms Branch and Bound Algorithm The Role of Algorithms in Computing Most important type of Algorithms DSA to DevelopmentCourse Branch and Bound Algorithm Last Updated : 22 Feb, 2024 The Branch and Bound Algorithm is a method 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 is commonly used in problems like the traveling salesman and job scheduling. Branch and Bound Algorithm Introduction to Branch and Bound: Branch and Bound meaning in DSA Introduction to Branch and Bound Difference between Backtracking and Branch-N-Bound Standard Problems on Branch and Bound: Branch and Bound | Set 1 (Introduction with 0/1 Knapsack) Branch and Bound | Set 2 (Implementation of 0/1 Knapsack) 0/1 Knapsack using Least Cost Branch and Bound Branch and Bound | Set 3 (8 puzzle Problem) Branch and Bound | Set 5 (N Queen Problem) Branch And Bound | Set 6 (Traveling Salesman Problem) Job Assignment Problem using Branch And Bound Travelling Salesman Problem (TSP) using Reduced Matrix Method Generate Binary Strings of length N using Branch and Bound Quick Links: Learn Data Structure and Algorithms | DSA Tutorial Comment More info Campus Training Program Next Article The Role of Algorithms in Computing course-img 641k+ interested Geeks DSA to Development: A Complete Guide Explore course-img 474k+ interested Geeks Complete Machine Learning & Data Science Program Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Here’s a concise summary of the **Branch and Bound Algorithm** based on the content you shared:

**Branch and Bound Algorithm — Overview**

**What is Branch and Bound?**

Branch and Bound (B&B) is a systematic method for solving **combinatorial optimization problems** by exploring branches of a search tree. It **divides** the problem into smaller subproblems (branches) and uses **bounds** to prune branches that cannot contain the optimal solution, thereby reducing the search space.

**How it works:**

1. **Branch**: Split the problem into smaller subproblems.
2. **Bound**: Calculate bounds on the best possible solution within a branch.
3. **Prune**: If a bound indicates that a branch cannot yield a better solution than the current best, discard (prune) that branch.
4. **Search**: Explore remaining branches until the optimal solution is found or all branches are exhausted.

**Key Characteristics**

* It guarantees finding the **optimal solution** (unlike greedy algorithms).
* Effective for problems where brute force would be too expensive.
* Often used with a **priority queue** or stack for managing nodes (subproblems).

**Typical Problems solved using Branch and Bound:**

* **0/1 Knapsack Problem**
* **8 Puzzle Problem**
* **N-Queen Problem**
* **Job Assignment Problem**
* **Travelling Salesman Problem (TSP)**

**Difference between Backtracking and Branch and Bound**

* **Backtracking** explores all feasible solutions and prunes branches that violate constraints.
* **Branch and Bound** uses bounds to prune branches that cannot improve on the current best solution, potentially pruning much earlier.

If you want, I can also help with detailed explanations or code examples of any specific Branch and Bound problem!

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... M DSA Course DSA Geometric Algorithms Basic Geometry Computational Geometry Slope of Line Point of Intersection of Two Lines Closest Pair of Points Convex Hull Pythagorean Quadruple Polygon Triangulation ▲ Explore GfG Courses Share Your Experiences Geometric Algorithms Problems based on Pattern Printing Problems based on Lines Problems based on Triangles Problems based on Rectangle, Square and Circle Problems based on 3D Objects Problems based on Quadrilateral Problems based on Polygon and Convex Hull Find Simple Closed Path for a given set of points Minimum Distance between Two Points Optimum location of point to minimize total distance Count of parallelograms in a plane Rotation of a point about another point Finding the vertex, focus and directrix of a parabola Find mirror image of a point in 2-D plane Equable Shapes DSA to DevelopmentCourse Geometric Algorithms Last Updated : 19 Jun, 2025 Geometric algorithms are a type of algorithm that deal with solving problems related to geometry. These algorithms are used to solve various geometric problems such as computing the area of a polygon, finding the intersection of geometric shapes, determining the convex hull of a set of points, and many other tasks related to geometric objects and their properties. Geometric algorithms are important in various fields such as computer graphics, computer-aided design, robotics, and geographical information systems. Pattern Printing: Print lower triangle with alternate ‘\*’ and ‘#’ Python Program to print the pattern ‘G’ Program to print pyramid pattern Program to print the diamond shape Hour-glass Pattern Program to print V and inverted-V pattern Program to print hollow pyramid, diamond pattern and their modifications Lines: Program to find the mid-point of a line Section formula (Point that divides a line in given ratio) Program to find slope of a line Program to find line passing through 2 Points Program for Point of Intersection of Two Lines Reflection of a point about a line in C++ Find points at a given distance on a line of given slope Given n line segments, find if any two segments intersect Count maximum points on same line Minimum lines to cover all points Triangle: Check whether triangle is valid or not if sides are given Check whether a given point lies inside a triangle or not Program to find area of a triangle Check if right triangle possible from given area and hypotenuse Count Integral points inside a Triangle Find all angles of a given triangle Program to find Circumcenter of a Triangle Number of Triangles that can be formed given a set of lines in Euclidean Plane Program to calculate area of Circumcircle of an Equilateral Triangle Program to find third side of triangle using law of cosines Rectangle | Square | Circle: Check whether a given point lies inside a rectangle or not Program for Area And Perimeter Of Rectangle Find if two rectangles overlap How to check if given four points form a square Number of rectangles in N\*M grid Program to find Circumference of a Circle Program to find area of a circle Check whether a point exists in circle sector or not Check if two given circles touch or intersect each other Program to find area of a Circular Segment Check if a line touches or intersects a circle Area of a Circumscribed Circle of a Square Find minimum radius such that atleast k point lie inside the circle Angular Sweep (Maximum points that can be enclosed in a circle of given radius) Quadrilateral: Number of parallelograms when n horizontal parallel lines intersect m vertical parallellines Program for Circumference of a Parallelogram Program to calculate area and perimeter of Trapezium Program to find area of a Trapezoid Find all possible coordinates of parallelogram Maximum area of quadrilateral Check whether four points make a parallelogram Find the Missing Point of Parallelogram 3D Objects: Find the perimeter of a cylinder Program for Volume and Surface area of Frustum of Cone Program to calculate volume of Ellipsoid Program for volume of Pyramid Calculate volume and surface area of a cone Calculate Volume and Surface area Of Sphere Program for Volume and Surface Area of Cuboid Program for Volume and Surface Area of Cube Pythagorean Quadruple LS3/NS3 sphere generation algorithm and its implementation Polygon and Convex Hull: How to check if a given point lies inside or outside a polygon? Area of a polygon with given n ordered vertices Tangents between two Convex Polygons Find number of diagonals in n sided convex polygon Convex Hull using Jarvis’ Algorithm or Wrapping Quickhull Algorithm for Convex Hull Deleting points from Convex Hull Minimum area of a Polygon with three points given Standard Problems on Geometric Algorithm: Finding the vertex, focus and directrix of a parabola Find Simple Closed Path for a given set of points Number of Integral Points between Two Points Optimum location of point to minimize total distance Find perimeter of shapes formed with 1s in binary matrix Draw geometric shapes on images using OpenCV Find if it’s possible to rotate the page by an angle or not Equable Shapes Quick Links : Learn Data Structure and Algorithms | DSA Tutorial ‘Practice Problems’ on Geometric Algorithms Recent Articles on Geometric Algorithms Comment More info Campus Training Program Next Article Print lower triangle with alternate '\*' and '#' course-img 1417k+ interested Geeks Data Structures and Algorithms - Self Paced [Online Course] Explore course-img 344k+ interested Geeks Data Structures & Algorithms in Python - Self Paced Explore course-img 112k+ interested Geeks Data Structures & Algorithms in JavaScript - Self Paced Course Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Here’s a clear overview of **Geometric Algorithms** based on the content you shared:

**Geometric Algorithms — Overview**

**What Are Geometric Algorithms?**

Geometric algorithms solve problems related to **geometry** — involving points, lines, shapes, polygons, and 3D objects. These algorithms calculate properties like area, intersections, distances, and more.

**Applications**

* **Computer graphics**
* **Robotics**
* **Geographical Information Systems (GIS)**
* **Computer-aided design (CAD)**
* **Pattern recognition**

**Common Topics & Problems**

**Pattern Printing**

* Triangles, pyramids, diamonds with characters

**Lines**

* Mid-point of a line
* Slope calculation
* Intersection of two lines
* Reflection of a point about a line
* Checking line segment intersections

**Triangles**

* Validity of triangles from side lengths
* Point inside triangle checks
* Area calculation
* Angles and circumcenters
* Number of triangles formed by given lines

**Rectangles, Squares, Circles**

* Point inside shapes
* Area and perimeter calculations
* Checking overlaps and intersections
* Circles: sector inclusion, circumference, radius calculations

**Quadrilaterals**

* Number and properties of parallelograms
* Area and perimeter calculations for trapeziums and trapezoids
* Identifying missing points

**3D Objects**

* Volume and surface area of cylinders, cones, spheres, cuboids, cubes, pyramids, ellipsoids

**Polygon and Convex Hull**

* Point inside polygon check
* Area of polygon
* Convex Hull algorithms (Jarvis’ March, Quickhull)
* Tangents and diagonals

**Example Problems**

* Optimum location to minimize total distance
* Finding closed paths in a set of points
* Counting integral points between points
* Drawing shapes on images (OpenCV)
* Rotation checks for shapes

If you want, I can explain any specific geometric algorithm or help with implementation examples in your preferred language!

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... M DSA Course DSA Tutorial on Randomized Algorithms Importance of Randomized Algorithms Random Variables Randomized Binary Search Algorithm Randomized Quick Sort Algorithm Las Vegas & Monte Carlo Algorithms Birthday Paradox Treap Skip List Generate Captcha Fisher-Yates Shuffle Reservoir Sampling Freivald's Algorithm 1/2 Approximate Median ▲ Explore GfG Courses Share Your Experiences Algorithms Tutorial What is an Algorithm | Introduction to Algorithms Definition, Types, Complexity and Examples of Algorithm Algorithms Design Techniques Why is Analysis of Algorithm important? Analysis of Algorithms Types of Algorithms Sorting Algorithms Searching Algorithms Greedy Algorithms Dynamic Programming or DP What is Pattern Searching ? Backtracking Algorithm Divide and Conquer Algorithm Mathematical Algorithms Geometric Algorithms Bitwise Algorithms Graph Algorithms Randomized Algorithms Branch and Bound Algorithm The Role of Algorithms in Computing Most important type of Algorithms DSA to DevelopmentCourse Randomized Algorithms Last Updated : 19 Jun, 2025 Randomized algorithms in data structures and algorithms (DSA) are algorithms that use randomness in their computations to achieve a desired outcome. These algorithms introduce randomness to improve efficiency or simplify the algorithm design. By incorporating random choices into their processes, randomized algorithms can often provide faster solutions or better approximations compared to deterministic algorithms. They are particularly useful in situations where exact solutions are difficult to find or when a probabilistic approach is acceptable. For example, in Randomized Quick Sort, we use a random number to pick the next pivot (or we randomly shuffle the array). We use randomization in Quick Sort to avoid a pattern for which it always causes a worst case. This is useful when we implement a library sort function using Quick Sort as it allows the library function to be a general purpose function and work almost uniformly for all cases. Introduction Introduction and Analysis Classification and Applications 1/2 Approximate Median Example Easy Problems Generates one of 3 according to given probabilities Generate 0 and 1 with 25% and 75% probability Implement rand3() using rand2() Birthday Paradox Expected Value of an array Shuffle a deck of cards Generate CAPTCHA and verify Index of max occurring with equal probability Randomized Binary Search Medium Problems Make a fair coin from a biased coin Shuffle a given array using Fisher–Yates shuffle Algorithm Expected Number of Trials until Success Strong Password Suggester Program QuickSort using Random Pivoting Operations on Sparse Matrices Estimating the value of Pi using Monte Carlo Implement rand12() using rand6() in one line Hard Problems Generate integer from 1 to 7 with equal probability Implement random-0-6-Generator using the given random-0-1-Generator Select a random number from stream, with O(1) space Random number generator in arbitrary probability distribution fashion Reservoir Sampling Linearity of Expectation Introduction and implementation of Karger’s algorithm for Minimum Cut Select a Random Node from a Singly Linked List Select a Random Node from a tree with equal probability Freivald’s Algorithm to check if a matrix is product of two Random Acyclic Maze Generator with given Entry and Exit point Recommended: Learn Data Structure and Algorithms | DSA Tutorial Comment More info Campus Training Program Next Article Branch and Bound Algorithm course-img 1418k+ interested Geeks Data Structures and Algorithms - Self Paced [Online Course] Explore course-img 344k+ interested Geeks Data Structures & Algorithms in Python - Self Paced Explore course-img 112k+ interested Geeks Data Structures & Algorithms in JavaScript - Self Paced Course Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Here’s a concise overview of **Randomized Algorithms** from the info you shared:

**Randomized Algorithms — Overview**

**What Are Randomized Algorithms?**

Randomized algorithms incorporate **randomness** into their logic or decision-making steps. They use random choices to improve efficiency, simplify design, or achieve probabilistic guarantees, often leading to faster or simpler solutions compared to deterministic approaches.

**Why Use Randomized Algorithms?**

* Avoid worst-case scenarios (e.g., Quick Sort’s pivot selection)
* Simplify algorithm design
* Achieve good average-case performance
* Useful where exact solutions are hard, and approximate or probabilistic answers are acceptable

**Examples of Randomized Algorithms**

* **Randomized Quick Sort:** Randomly select pivot to avoid worst-case sorting time.
* **Randomized Binary Search:** Randomly pick points for search.
* **Fisher-Yates Shuffle:** Efficiently shuffle an array or deck of cards uniformly at random.
* **Reservoir Sampling:** Randomly sample elements from a stream without knowing its length upfront.
* **Freivald’s Algorithm:** Probabilistic algorithm to verify matrix multiplication.

**Key Concepts & Topics**

* Random Variables
* Las Vegas vs Monte Carlo algorithms
* Birthday Paradox
* Generating random numbers with specific probabilities (e.g., rand3() from rand2())
* Expected value calculations
* Approximate medians
* Probabilistic data structures like Treaps and Skip Lists

**Example Applications**

* Generating CAPTCHA codes
* Password strength suggestion
* Estimating π using Monte Carlo methods
* Random node selection from data structures like linked lists or trees
* Random maze generation

If you'd like, I can help explain any particular randomized algorithm in detail or provide coding examples!

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... 17 M DSA Course DSA Tutorial Data Structures Algorithms Array Strings Linked List Stack Queue Tree Graph Searching Sorting Recursion Dynamic Programming Binary Tree Binary Search Tree Heap Hashing Divide & Conquer Mathematical Geometric Bitwise Greedy Backtracking Branch and Bound Matrix Pattern Searching Randomized ▲ Explore GfG Courses Share Your Experiences Advanced Data Structures Generic Linked List in C Memory efficient doubly linked list XOR Linked List - A Memory Efficient Doubly Linked List | Set 1 XOR Linked List – A Memory Efficient Doubly Linked List | Set 2 Skip List - Efficient Search, Insert and Delete in Linked List Self Organizing List | Set 1 (Introduction) Unrolled Linked List | Set 1 (Introduction) Splay Tree Trie Trie Data Structure Tutorial Segment Tree Scape Goat Tree and Treap DSA to DevelopmentCourse Trie Data Structure Tutorial Last Updated : 21 Apr, 2025 The trie data structure, also known as a prefix tree, is a tree-like data structure used for efficient retrieval of key-value pairs. It is commonly used for implementing dictionaries and autocomplete features, making it a fundamental component in many search algorithms. In this article, we will explore all about Trie data structures in detail. trie-data Trie Data Structure Table of Content What is Trie Data Structure? What is need of Trie Data Structure? Advantages of Trie Data Structure over a Hash Table Properties of a Trie Data Structure How does Trie Data Structure work? Representation of Trie Node Basic Operations on Trie Data Structure Insertion in Trie Data Structure Searching in Trie Data Structure Deletion in Trie Data Structure Implement Trie Data Structure? Complexity Analysis of Trie Data Structure Applications of Trie data structure Advantages of Trie data structure Disadvantages of Trie data structure Top Interview problems on Trie data structure What is Trie Data Structure? Trie data structure is defined as a Tree based data structure that is used for storing a collection of strings and performing efficient search, insert, delete, prefix search and sorted-traversal-of-all operations on them. The word Trie is derived from reTRIEval, which means finding something or obtaining it. Trie data structure follows a property that If two strings have a common prefix then they will have the same ancestor in the trie. This particular property allows to find all words with a given prefix. What is need of Trie Data Structure? A Trie data structure is used for storing and retrieval of data and the same operations could be done using another data structure which is Hash Table but Trie data structure can perform these operations more efficiently than a Hash Table. Moreover, Trie has its own advantage over the Hash table. A Trie data structure can be used for prefix-based searching and a sorted traversal of all words. So a Trie has advantages of both hash table and self balancing binary search trees. However the main issue with Trie is extra memory space required to store words and the space may become huge for long list of words and/or for long words. Advantages of Trie Data Structure over a Hash Table: The A trie data structure has the following advantages over a hash table: We can efficiently do prefix search (or auto-complete) with Trie. We can easily print all words in alphabetical order which is not easily possible with hashing. There is no overhead of Hash functions in a Trie data structure. Searching for a String even in the large collection of strings in a Trie data structure can be done in O(L) Time complexity, Where L is the number of words in the query string. This searching time could be even less than O(L) if the query string does not exist in the trie. Properties of a Trie Data Structure Below are some important properties of the Trie data structure: Each Trie has an empty root node, with links (or references) to other nodes Each node of a Trie represents a string and each edge represents a character. Every node consists of hashmaps or an array of pointers, with each index representing a character and a flag to indicate if any string ends at the current node. Trie data structure can contain any number of characters including alphabets, numbers, and special characters. But for this article, we will discuss strings with characters a-z. Therefore, only 26 pointers need for every node, where the 0th index represents 'a' and the 25th index represents 'z' characters. Each path from the root to any node represents a word or string. Below is a simple example of Trie data structure. Trie Data Structure Trie Data Structure How does Trie Data Structure work? Trie data structure can contain any number of characters including alphabets, numbers, and special characters. But for this article, we will discuss strings with characters a-z. Therefore, only 26 pointers need for every node, where the 0th index represents 'a' and the 25th index represents 'z' characters. Any lowercase English word can start with a-z, then the next letter of the word could be a-z, the third letter of the word again could be a-z, and so on. So for storing a word, we need to take an array (container) of size 26 and initially, all the characters are empty as there are no words and it will look as shown below. An array of pointers inside every Trie node An array of pointers inside every Trie node Let's see how a word "and" and "ant" is stored in the Trie data structure: Store "and" in Trie data structure: The word "and" starts with "a", So we will mark the position "a" as filled in the Trie node, which represents the use of "a". After placing the first character, for the second character again there are 26 possibilities, So from "a", again there is an array of size 26, for storing the 2nd character. The second character is "n", So from "a", we will move to "n" and mark "n" in the 2nd array as used. After "n", the 3rd character is "d", So mark the position "d" as used in the respective array. Store "ant" in the Trie data structure: The word "ant" starts with "a" and the position of "a" in the root node has already been filled. So, no need to fill it again, just move to the node 'a' in Trie. For the second character 'n' we can observe that the position of 'n' in the 'a' node has already been filled. So, no need to fill it again, just move to node 'n' in Trie. For the last character 't' of the word, The position for 't' in the 'n' node is not filled. So, filled the position of 't' in 'n' node and move to 't' node. After storing the word "and" and "ant" the Trie will look like this: Representation of Trie Node: Every Trie node consists of a character pointer array or hashmap and a flag to represent if the word is ending at that node or not. But if the words contain only lower-case letters (i.e. a-z), then we can define Trie Node with an array instead of a hashmap. Try it on GfG Practice redirect icon struct TrieNode { struct TrieNode\* children[ALPHABET\_SIZE]; // This will keep track of number of strings that are // stored in the Trie from root node to any Trie node. int wordCount = 0; }; Basic Operations on Trie Data Structure: Insertion Search Deletion 1. Insertion in Trie Data Structure: This operation is used to insert new strings into the Trie data structure. Let us see how this works: Let us try to Insert "and" & "ant" in this Trie: Insert "and" & "ant" Insert "and" & "ant" From the above representation of insertion, we can see that the word "and" & "ant" have shared some common node (i.e "an") this is because of the property of the Trie data structure that If two strings have a common prefix then they will have the same ancestor in the trie. Now let us try to Insert "dad" & "do": Insertion in Trie Data Structure Insertion in Trie Data Structure Implementation of Insertion in Trie data structure: Algorithm: Define a function insert(TrieNode \*root, string &word) which will take two parameters one for the root and the other for the string that we want to insert in the Trie data structure. Now take another pointer currentNode and initialize it with the root node. Iterate over the length of the given string and check if the value is NULL or not in the array of pointers at the current character of the string. If It's NULL then, make a new node and point the current character to this newly created node. Move the curr to the newly created node. Finally, increment the wordCount of the last currentNode, this implies that there is a string ending currentNode. Below is the implementation of the above algorithm: void insert\_key(TrieNode\* root, string& key) { // Initialize the currentNode pointer // with the root node TrieNode\* currentNode = root; // Iterate across the length of the string for (auto c : key) { // Check if the node exist for the current // character in the Trie. if (currentNode->childNode[c - 'a'] == NULL) { // If node for current character does not exist // then make a new node TrieNode\* newNode = new TrieNode(); // Keep the reference for the newly created // node. currentNode->childNode[c - 'a'] = newNode; } // Now, move the current node pointer to the newly // created node. currentNode = currentNode->childNode[c - 'a']; } // Increment the wordEndCount for the last currentNode // pointer this implies that there is a string ending at // currentNode. currentNode->wordCount++; } 2. Searching in Trie Data Structure: Search operation in Trie is performed in a similar way as the insertion operation but the only difference is that whenever we find that the array of pointers in curr node does not point to the current character of the word then return false instead of creating a new node for that current character of the word. This operation is used to search whether a string is present in the Trie data structure or not. There are two search approaches in the Trie data structure. Find whether the given word exists in Trie. Find whether any word that starts with the given prefix exists in Trie. There is a similar search pattern in both approaches. The first step in searching a given word in Trie is to convert the word to characters and then compare every character with the trie node from the root node. If the current character is present in the node, move forward to its children. Repeat this process until all characters are found. 2.1 Searching Prefix in Trie Data Structure: Search for the prefix "an" in the Trie Data Structure. Search for the prefix "an" in Trie Search for the prefix "an" in Trie Implementation of Prefix Search in Trie data structure: bool isPrefixExist(TrieNode\* root, string& key) { // Initialize the currentNode pointer // with the root node TrieNode\* currentNode = root; // Iterate across the length of the string for (auto c : key) { // Check if the node exist for the current // character in the Trie. if (currentNode->childNode[c - 'a'] == NULL) { // Given word as a prefix does not exist in Trie return false; } // Move the currentNode pointer to the already // existing node for current character. currentNode = currentNode->childNode[c - 'a']; } // Prefix exist in the Trie return true; } 2.2 Searching Complete word in Trie Data Structure: It is similar to prefix search but additionally, we have to check if the word is ending at the last character of the word or not. Searching in Trie Data Structure Search "dad" in the Trie data structure Implementation of Search in Trie data structure: bool search\_key(TrieNode\* root, string& key) { // Initialize the currentNode pointer // with the root node TrieNode\* currentNode = root; // Iterate across the length of the string for (auto c : key) { // Check if the node exist for the current // character in the Trie. if (currentNode->childNode[c - 'a'] == NULL) { // Given word does not exist in Trie return false; } // Move the currentNode pointer to the already // existing node for current character. currentNode = currentNode->childNode[c - 'a']; } return (currentNode->wordCount > 0); } 3. Deletion in Trie Data Structure This operation is used to delete strings from the Trie data structure. There are three cases when deleting a word from Trie. The deleted word is a prefix of other words in Trie. The deleted word shares a common prefix with other words in Trie. The deleted word does not share any common prefix with other words in Trie. 3.1 The deleted word is a prefix of other words in Trie. As shown in the following figure, the deleted word "an" share a complete prefix with another word "and" and "ant". Deletion of word which is a prefix of other words in Trie Deletion of word which is a prefix of other words in Trie An easy solution to perform a delete operation for this case is to just decrement the wordCount by 1 at the ending node of the word. 3.2 The deleted word shares a common prefix with other words in Trie. As shown in the following figure, the deleted word "and" has some common prefixes with other words ‘ant’. They share the prefix ‘an’. Deletion of word which shares a common prefix with other words in Trie Deletion of word which shares a common prefix with other words in Trie The solution for this case is to delete all the nodes starting from the end of the prefix to the last character of the given word. 3.3 The deleted word does not share any common prefix with other words in Trie. As shown in the following figure, the word "geek" does not share any common prefix with any other words. The solution for this case is just to delete all the nodes. Below is the implementation that handles all the above cases: bool delete\_key(TrieNode\* root, string& word) { TrieNode\* currentNode = root; TrieNode\* lastBranchNode = NULL; char lastBranchChar = 'a'; for (auto c : word) { if (currentNode->childNode[c - 'a'] == NULL) { return false; } else { int count = 0; for (int i = 0; i < 26; i++) { if (currentNode->childNode[i] != NULL) count++; } if (count > 1) { lastBranchNode = currentNode; lastBranchChar = c; } currentNode = currentNode->childNode[c - 'a']; } } int count = 0; for (int i = 0; i < 26; i++) { if (currentNode->childNode[i] != NULL) count++; } // Case 1: The deleted word is a prefix of other words // in Trie. if (count > 0) { currentNode->wordCount--; return true; } // Case 2: The deleted word shares a common prefix with // other words in Trie. if (lastBranchNode != NULL) { lastBranchNode->childNode[lastBranchChar] = NULL; return true; } // Case 3: The deleted word does not share any common // prefix with other words in Trie. else { root->childNode[word[0]] = NULL; return true; } } Implement Trie Data Structure? Algorithm: Create a root node with the help of TrieNode() constructor. Store a collection of strings that we have to insert in the trie in a vector of strings say, arr. Inserting all strings in Trie with the help of the insertkey() function, Search strings from searchQueryStrings with the help of search\_key() function. Delete the strings present in the deleteQueryStrings with the help of delete\_key. #include <bits/stdc++.h> using namespace std; ​ struct TrieNode { ​ // pointer array for child nodes of each node TrieNode\* childNode[26]; int wordCount; ​ TrieNode() { // constructor // initialize the wordCnt variable with 0 // initialize every index of childNode array with // NULL wordCount = 0; for (int i = 0; i < 26; i++) { childNode[i] = NULL; } } }; ​ void insert\_key(TrieNode\* root, string& key) { // Initialize the currentNode pointer // with the root node TrieNode\* currentNode = root; ​ // Iterate across the length of the string for (auto c : key) { ​ // Check if the node exist for the current // character in the Trie. if (currentNode->childNode[c - 'a'] == NULL) { ​ // If node for current character does not exist // then make a new node TrieNode\* newNode = new TrieNode(); ​ // Keep the reference for the newly created // node. currentNode->childNode[c - 'a'] = newNode; } ​ // Now, move the current node pointer to the newly // created node. currentNode = currentNode->childNode[c - 'a']; } ​ // Increment the wordEndCount for the last currentNode // pointer this implies that there is a string ending at // currentNode. currentNode->wordCount++; } ​ bool search\_key(TrieNode\* root, string& key) { // Initialize the currentNode pointer // with the root node TrieNode\* currentNode = root; ​ // Iterate across the length of the string for (auto c : key) { ​ // Check if the node exist for the current // character in the Trie. if (currentNode->childNode[c - 'a'] == NULL) { ​ // Given word does not exist in Trie return false; } ​ // Move the currentNode pointer to the already // existing node for current character. currentNode = currentNode->childNode[c - 'a']; } ​ return (currentNode->wordCount > 0); } ​ bool delete\_key(TrieNode\* root, string& word) { TrieNode\* currentNode = root; TrieNode\* lastBranchNode = NULL; char lastBrachChar = 'a'; ​ for (auto c : word) { if (currentNode->childNode[c - 'a'] == NULL) { return false; } else { int count = 0; for (int i = 0; i < 26; i++) { if (currentNode->childNode[i] != NULL) count++; } ​ if (count > 1) { lastBranchNode = currentNode; lastBrachChar = c; } currentNode = currentNode->childNode[c - 'a']; } } ​ int count = 0; for (int i = 0; i < 26; i++) { if (currentNode->childNode[i] != NULL) count++; } ​ // Case 1: The deleted word is a prefix of other words // in Trie. if (count > 0) { currentNode->wordCount--; return true; } ​ // Case 2: The deleted word shares a common prefix with // other words in Trie. if (lastBranchNode != NULL) { lastBranchNode->childNode[lastBrachChar] = NULL; return true; } // Case 3: The deleted word does not share any common // prefix with other words in Trie. else { root->childNode[word[0]] = NULL; return true; } } ​ // Driver code int main() { // Make a root node for the Trie TrieNode\* root = new TrieNode(); ​ // Stores the strings that we want to insert in the // Trie vector<string> inputStrings = { "and", "ant", "do", "geek", "dad", "ball" }; ​ // number of insert operations in the Trie int n = inputStrings.size(); ​ for (int i = 0; i < n; i++) { insert\_key(root, inputStrings[i]); } ​ // Stores the strings that we want to search in the Trie vector<string> searchQueryStrings = { "do", "geek", "bat" }; ​ // number of search operations in the Trie int searchQueries = searchQueryStrings.size(); ​ for (int i = 0; i < searchQueries; i++) { cout << "Query String: " << searchQueryStrings[i] << "\n"; if (search\_key(root, searchQueryStrings[i])) { // the queryString is present in the Trie cout << "The query string is present in the " "Trie\n"; } else { // the queryString is not present in the Trie cout << "The query string is not present in " "the Trie\n"; } } ​ // stores the strings that we want to delete from the // Trie vector<string> deleteQueryStrings = { "geek", "tea" }; ​ // number of delete operations from the Trie int deleteQueries = deleteQueryStrings.size(); ​ for (int i = 0; i < deleteQueries; i++) { cout << "Query String: " << deleteQueryStrings[i] << "\n"; if (delete\_key(root, deleteQueryStrings[i])) { // The queryString is successfully deleted from // the Trie cout << "The query string is successfully " "deleted\n"; } else { // The query string is not present in the Trie cout << "The query string is not present in " "the Trie\n"; } } ​ return 0; } Output Query String: do The query string is present in the Trie Query String: geek The query string is present in the Trie Query String: bat The query string is not present in the Trie Query String: geek The query string is successfully deleted Query String: tea The query string is not present in the Trie Complexity Analysis of Trie Data Structure Operation Time Complexity Insertion O(n) Here n is the length of string to be searched Searching O(n) Deletion O(n) Note: In the above complexity table 'n', 'm' represents the size of the string and the number of strings that are stored in the trie. Applications of Trie data structure: 1. Autocomplete Feature: Autocomplete provides suggestions based on what you type in the search box. Trie data structure is used to implement autocomplete functionality. Autocomplete feature of Trie Data Structure Autocomplete feature of Trie Data Structure 2. Spell Checkers: If the word typed does not appear in the dictionary, then it shows suggestions based on what you typed. It is a 3-step process that includes : Checking for the word in the data dictionary. Generating potential suggestions. Sorting the suggestions with higher priority on top. Trie stores the data dictionary and makes it easier to build an algorithm for searching the word from the dictionary and provides the list of valid words for the suggestion. 3. Longest Prefix Matching Algorithm(Maximum Prefix Length Match): This algorithm is used in networking by the routing devices in IP networking. Optimization of network routes requires contiguous masking that bound the complexity of lookup a time to O(n), where n is the length of the URL address in bits. To speed up the lookup process, Multiple Bit trie schemes were developed that perform the lookups of multiple bits faster. Advantages of Trie data structure: Trie allows us to input and finds words in O(n) time, where n is the length of a single word. It is faster as compared to both hash tables and binary search trees. It provides alphabetical filtering of entries by the key of the node and hence makes it easier to print all words in alphabetical order. Prefix search/Longest prefix matching can be efficiently done with the help of trie data structure. Since trie doesn’t need any hash function for its implementation so they are generally faster than hash tables for small keys like integers and pointers. Tries support ordered iteration whereas iteration in a hash table will result in pseudorandom order given by the hash function which is usually more cumbersome. Deletion is also a straightforward algorithm with O(n) as its time complexity, where n is the length of the word to be deleted. Disadvantages of Trie data structure: The main disadvantage of the trie is that it takes a lot of memory to store all the strings. For each node, we have too many node pointers which are equal to the no of characters in the worst case. An efficiently constructed hash table(i.e. a good hash function and a reasonable load factor) has O(1) as lookup time which is way faster than O(l) in the case of a trie, where l is the length of the string. Top Interview problems on Trie data structure: S.no Problem Practice 1 Implement Trie (Prefix Tree) Link 2 Word Break Problem Link 3 Boggle Link 4 Longest Common Prefix using Trie Link 5 Find the maximum subarray XOR in a given array Link 6 Count of distinct substrings of a string Link 7 Find shortest unique prefix for every word in a given list Link 8 Count inversions in an array Link Conclusion: Our discussion so far has led us to the conclusion that the Trie data structure is a Tree based data structure that is used for storing some collection of strings and performing efficient search operations on them and we have also discussed the various advantage and applications of trie data structure. Related articles: How to start data learning DSA? Competitive Programming – A Complete Guide How can one become good at Data Structures and Algorithms easily? Why Data Structures and Algorithms Are Important to Learn? SDE SHEET – A Complete Guide for SDE Preparation Google Interview Preparation For Software Engineer – A Complete Guide 100 Days of Code – A Complete Guide For Beginners and Experienced Introduction to Trie Data Structure Visit Course explore course icon Comment More info Campus Training Program Next Article Segment Tree course-img 640k+ interested Geeks DSA to Development: A Complete Guide Explore course-img 474k+ interested Geeks Complete Machine Learning & Data Science Program Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Here’s a clear, concise summary of the **Trie Data Structure** based on the detailed content you shared:

**Trie Data Structure (Prefix Tree)**

**What is a Trie?**

* A **Trie** is a tree-like data structure used to store a collection of strings.
* It enables **efficient insertion, search, deletion, prefix search, and sorted traversal** of strings.
* Name "Trie" comes from **reTRIEval** — meaning finding something efficiently.
* Commonly used for autocomplete, spell checking, and prefix-based queries.

**Why Use a Trie?**

* Performs **prefix search** efficiently (not possible directly with hash tables).
* Can print words in **alphabetical order** easily.
* Searching time depends on length of the query string, **O(L)** where L = length of the string.
* No overhead of hash functions (unlike hash tables).
* Useful for applications where **prefix matching** is important (autocomplete, spell checkers, IP routing).

**Structure & Properties**

* Each node has an array (or hashmap) of child pointers (typically size 26 for a-z).
* Each path from root to a node represents a prefix or whole string.
* Nodes contain a flag or counter indicating if a word ends there.
* Words sharing prefixes share ancestor nodes in the trie.

**Basic Operations**

1. **Insertion:** Insert characters one-by-one; create new nodes if missing.
2. **Search:** Traverse nodes matching characters; return true if word ends here.
3. **Prefix Search:** Like search but only checks if prefix exists.
4. **Deletion:** Remove nodes carefully considering three cases:
   * Word is prefix of other words (just decrement count).
   * Word shares prefix with others (remove nodes after branching).
   * Word has no common prefix (delete entire branch).

**Example Code Snippet (C++)**

struct TrieNode { TrieNode\* childNode[26]; int wordCount; TrieNode() { wordCount = 0; for (int i = 0; i < 26; i++) childNode[i] = NULL; } }; void insert\_key(TrieNode\* root, string& key) { TrieNode\* currentNode = root; for (char c : key) { if (currentNode->childNode[c - 'a'] == NULL) currentNode->childNode[c - 'a'] = new TrieNode(); currentNode = currentNode->childNode[c - 'a']; } currentNode->wordCount++; } bool search\_key(TrieNode\* root, string& key) { TrieNode\* currentNode = root; for (char c : key) { if (currentNode->childNode[c - 'a'] == NULL) return false; currentNode = currentNode->childNode[c - 'a']; } return (currentNode->wordCount > 0); }

**Time Complexity**

| **Operation** | **Time Complexity** |
| --- | --- |
| Insertion | O(n), n = length of word |
| Search | O(n) |
| Deletion | O(n) |

**Applications**

* **Autocomplete:** Suggest words based on typed prefix.
* **Spell Checking:** Find words and suggest corrections.
* **Longest Prefix Matching:** Used in networking (IP routing).
* **Dictionary implementations**

**Advantages**

* Fast prefix search and word lookup.
* Alphabetical ordering of words is straightforward.
* No hash collisions or overhead.
* Supports ordered iteration.
* Straightforward deletion.

**Disadvantages**

* High memory usage (each node stores pointers for all characters).
* For very large datasets, tries can consume large amounts of space.
* Hash tables have O(1) average lookup and can be faster for some lookups.

If you want, I can help you with implementing specific operations or solving trie-based problems. Just ask!

**You said:**

Skip to content geeksforgeeks Courses Tutorials Practice Jobs Search... DSA Course DSA Tutorial Data Structures Algorithms Array Strings Linked List Stack Queue Tree Graph Searching Sorting Recursion Dynamic Programming Binary Tree Binary Search Tree Heap Hashing Divide & Conquer Mathematical Geometric Bitwise Greedy Backtracking Branch and Bound Matrix Pattern Searching Randomized Explore GfG Courses Share Your Experiences Introduction to Red-Black Tree Red-Black Tree definition & meaning in DSA Applications, Advantages and Disadvantages of Red-Black Tree Insertion in Red-Black Tree C Program for Red Black Tree Insertion Deletion in Red-Black Tree Red-Black Trees | Top-Down Insertion Left Leaning Red Black Tree (Insertion) Check if a given Binary Tree is height balanced like a Red-Black Tree What is the difference between Heap and Red-Black Tree? Red Black Tree vs AVL Tree DSA to DevelopmentCourse Introduction to Red-Black Tree Last Updated : 25 Apr, 2025 Binary search trees are a fundamental data structure, but their performance can suffer if the tree becomes unbalanced. Red Black Trees are a type of balanced binary search tree that use a set of rules to maintain balance, ensuring logarithmic time complexity for operations like insertion, deletion, and searching, regardless of the initial shape of the tree. Red Black Trees are self-balancing, using a simple color-coding scheme to adjust the tree after each modification. Red-black-Tree-banner Red-Black Tree Table of Content What is a Red-Black Tree? Properties of Red-Black Trees Example of Red-Black Tree Why Red-Black Trees? Comparison with AVL Tree: Interesting points about Red-Black Tree: Basic Operations on Red-Black Tree: 1. Insertion 2. Searching 3. Deletion 4. Rotation Left Rotation Right Rotation When to Perform Rotations? Implementation of Red-Black Tree: Advantages of Red-Black Trees: Disadvantages of Red-Black Trees: Applications of Red-Black Trees: What is a Red-Black Tree? A Red-Black Tree is a self-balancing binary search tree where each node has an additional attribute: a color, which can be either red or black. The primary objective of these trees is to maintain balance during insertions and deletions, ensuring efficient data retrieval and manipulation. Properties of Red-Black Trees A Red-Black Tree have the following properties: Node Color: Each node is either red or black. Root Property: The root of the tree is always black. Red Property: Red nodes cannot have red children (no two consecutive red nodes on any path). Black Property: Every path from a node to its descendant null nodes (leaves) has the same number of black nodes. Leaf Property: All leaves (NIL nodes) are black. These properties ensure that the longest path from the root to any leaf is no more than twice as long as the shortest path, maintaining the tree's balance and efficient performance. Example of Red-Black Tree: New-Project-8 The Correct Red-Black Tree in above image ensures that every path from the root to a leaf node has the same number of black nodes. In this case,​ there is one (excluding the root node). The Incorrect Red Black Tree does not follow the red-black properties as two red nodes are adjacent to each other. Another problem is that one of the paths to a leaf node has zero black nodes, whereas the other two contain a black node. Why Red-Black Trees? Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that the height of the tree remains O(log n) after every insertion and deletion, then we can guarantee an upper bound of O(log n) for all these operations. The height of a Red-Black tree is always O(log n) where n is the number of nodes in the tree. Sr. No. Algorithm Time Complexity 1. Search O(log n) 2. Insert O(log n) 3. Delete O(log n) Comparison with AVL Tree: The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves frequent insertions and deletions, then Red-Black trees should be preferred. And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over the Red-Black Tree. How does a Red-Black Tree ensure balance? A simple example to understand balancing is, that a chain of 3 nodes is not possible in the Red-Black tree. We can try any combination of colors and see if all of them violate the Red-Black tree property. Proper structure of three noded Red-black tree Interesting points about Red-Black Tree: The black height of the red-black tree is the number of black nodes on a path from the root node to a leaf node. Leaf nodes are also counted as black nodes. So, a red-black tree of height h has black height >= h/2. Height of a red-black tree with n nodes is h<= 2 log2(n + 1). All leaves (NIL) are black. The black depth of a node is defined as the number of black nodes from the root to that node i.e the number of black ancestors. Basic Operations on Red-Black Tree: The basic operations on a Red-Black Tree include: Insertion Search Deletion Rotation 1. Insertion Inserting a new node in a Red-Black Tree involves a two-step process: performing a standard binary search tree (BST) insertion, followed by fixing any violations of Red-Black properties. Insertion Steps BST Insert: Insert the new node like in a standard BST. Fix Violations: If the parent of the new node is black, no properties are violated. If the parent is red, the tree might violate the Red Property, requiring fixes. Fixing Violations During Insertion After inserting the new node as a red node, we might encounter several cases depending on the colors of the node's parent and uncle (the sibling of the parent): Case 1: Uncle is Red: Recolor the parent and uncle to black, and the grandparent to red. Then move up the tree to check for further violations. Case 2: Uncle is Black: Sub-case 2.1: Node is a right child: Perform a left rotation on the parent. Sub-case 2.2: Node is a left child: Perform a right rotation on the grandparent and recolor appropriately. 2. Searching Searching for a node in a Red-Black Tree is similar to searching in a standard Binary Search Tree (BST). The search operation follows a straightforward path from the root to a leaf, comparing the target value with the current node's value and moving left or right accordingly. Search Steps Start at the Root: Begin the search at the root node. Traverse the Tree: If the target value is equal to the current node's value, the node is found. If the target value is less than the current node's value, move to the left child. If the target value is greater than the current node's value, move to the right child. Repeat: Continue this process until the target value is found or a NIL node is reached (indicating the value is not present in the tree). 3. Deletion Deleting a node from a Red-Black Tree also involves a two-step process: performing the BST deletion, followed by fixing any violations that arise. Deletion Steps BST Deletion: Remove the node using standard BST rules. Fix Double Black: If a black node is deleted, a "double black" condition might arise, which requires specific fixes. Fixing Violations During Deletion When a black node is deleted, we handle the double black issue based on the sibling's color and the colors of its children: Case 1: Sibling is Red: Rotate the parent and recolor the sibling and parent. Case 2: Sibling is Black: Sub-case 2.1: Sibling's children are black: Recolor the sibling and propagate the double black upwards. Sub-case 2.2: At least one of the sibling's children is red: If the sibling's far child is red: Perform a rotation on the parent and sibling, and recolor appropriately. If the sibling's near child is red: Rotate the sibling and its child, then handle as above. 4. Rotation Rotations are fundamental operations in maintaining the balanced structure of a Red-Black Tree (RBT). They help to preserve the properties of the tree, ensuring that the longest path from the root to any leaf is no more than twice the length of the shortest path. Rotations come in two types: left rotations and right rotations. 1. Left Rotation A left rotation at node 𝑥x moves 𝑥x down to the left and its right child 𝑦y up to take 𝑥x’s place. Before Rotation: x \ y / \ a b After Left Rotation: y / \ x b \ a Left Rotation Steps: Set y to be the right child of x. Move y’s left subtree to x’s right subtree. Update the parent of x and y. Update x’s parent to point to y instead of x. Set y’s left child to x. Update x’s parent to y. Pseudocode of Left Rotation: // Utility function to perform left rotation void leftRotate(Node\* x) { Node\* y = x->right; x->right = y->left; if (y->left != NIL) { y->left->parent = x; } y->parent = x->parent; if (x->parent == nullptr) { root = y; } else if (x == x->parent->left) { x->parent->left = y; } else { x->parent->right = y; } y->left = x; x->parent = y; } 2. Right Rotation A right rotation at node 𝑥x moves 𝑥x down to the right and its left child 𝑦y up to take 𝑥x’s place. Befor Right Rotation: ​ x / y / \ a b ​ After Right Rotation: ​ y / \ a x / b Right Rotation Steps: Set y to be the left child of x. Move y’s right subtree to x’s left subtree. Update the parent of x and y. Update x’s parent to point to y instead of x. Set y’s right child to x. Update x’s parent to y. Pseudocode of Right Rotation: // Utility function to perform right rotation void rightRotate(Node\* x) { Node\* y = x->left; x->left = y->right; if (y->right != NIL) { y->right->parent = x; } y->parent = x->parent; if (x->parent == nullptr) { root = y; } else if (x == x->parent->right) { x->parent->right = y; } else { x->parent->left = y; } y->right = x; x->parent = y; } When to Perform Rotations? Rotations in Red-Black Trees are typically performed during insertions and deletions to maintain the properties of the tree. Below are the scenarios for rotations: 1. Fixing Violations after Insertion When a new node is inserted, it is always colored red. This can create violations of Red-Black Tree properties, specifically: The root must be black. Red nodes cannot have red children. Case Analysis for Fixing Insertions: Case 1: Recoloring and Propagating Upwards If the parent and uncle of the new node are both red, recolor the parent and uncle to black, and the grandparent to red. Then, recursively apply the fix-up to the grandparent. Case 2: Rotation and Recoloring If the new node’s uncle is black and the new node is the right child of a left child (or vice versa), perform a rotation to move the new node up and align it. If the new node’s uncle is black and the new node is the left child of a left child (or right of a right), perform a rotation and recolor the parent and grandparent to fix the violation. 2. Fixing Violations after Deletion After deletion, the tree might need fixing to restore properties: When a black node is removed, or a red node is replaced by a black node, a double-black situation can arise. Case Analysis for Fixing Deletions: Case 1: Sibling is Red Recolor the sibling and the parent, and perform a rotation. Case 2: Sibling is Black with Black Children Recolor the sibling to red and move the problem up to the parent. Case 3: Sibling is Black with at least one Red Child Rotate and recolor to fix the double-black issue. Implementation of Red-Black Tree: Here's a detailed implementation of a Red-Black Tree including insertion, search, and rotation functions: #include <iostream> using namespace std; ​ // Node structure for the Red-Black Tree struct Node { int data; string color; Node \*left, \*right, \*parent; ​ Node(int data) : data(data) , color("RED") , left(nullptr) , right(nullptr) , parent(nullptr) { } }; ​ // Red-Black Tree class class RedBlackTree { private: Node\* root; Node\* NIL; ​ // Utility function to perform left rotation void leftRotate(Node\* x) { Node\* y = x->right; x->right = y->left; if (y->left != NIL) { y->left->parent = x; } y->parent = x->parent; if (x->parent == nullptr) { root = y; } else if (x == x->parent->left) { x->parent->left = y; } else { x->parent->right = y; } y->left = x; x->parent = y; } ​ // Utility function to perform right rotation void rightRotate(Node\* x) { Node\* y = x->left; x->left = y->right; if (y->right != NIL) { y->right->parent = x; } y->parent = x->parent; if (x->parent == nullptr) { root = y; } else if (x == x->parent->right) { x->parent->right = y; } else { x->parent->left = y; } y->right = x; x->parent = y; } ​ // Function to fix Red-Black Tree properties after // insertion void fixInsert(Node\* k) { while (k != root && k->parent->color == "RED") { if (k->parent == k->parent->parent->left) { Node\* u = k->parent->parent->right; // uncle if (u->color == "RED") { k->parent->color = "BLACK"; u->color = "BLACK"; k->parent->parent->color = "RED"; k = k->parent->parent; } else { if (k == k->parent->right) { k = k->parent; leftRotate(k); } k->parent->color = "BLACK"; k->parent->parent->color = "RED"; rightRotate(k->parent->parent); } } else { Node\* u = k->parent->parent->left; // uncle if (u->color == "RED") { k->parent->color = "BLACK"; u->color = "BLACK"; k->parent->parent->color = "RED"; k = k->parent->parent; } else { if (k == k->parent->left) { k = k->parent; rightRotate(k); } k->parent->color = "BLACK"; k->parent->parent->color = "RED"; leftRotate(k->parent->parent); } } } root->color = "BLACK"; } ​ // Inorder traversal helper function void inorderHelper(Node\* node) { if (node != NIL) { inorderHelper(node->left); cout << node->data << " "; inorderHelper(node->right); } } ​ // Search helper function Node\* searchHelper(Node\* node, int data) { if (node == NIL || data == node->data) { return node; } if (data < node->data) { return searchHelper(node->left, data); } return searchHelper(node->right, data); } ​ public: // Constructor RedBlackTree() { NIL = new Node(0); NIL->color = "BLACK"; NIL->left = NIL->right = NIL; root = NIL; } ​ // Insert function void insert(int data) { Node\* new\_node = new Node(data); new\_node->left = NIL; new\_node->right = NIL; ​ Node\* parent = nullptr; Node\* current = root; ​ // BST insert while (current != NIL) { parent = current; if (new\_node->data < current->data) { current = current->left; } else { current = current->right; } } ​ new\_node->parent = parent; ​ if (parent == nullptr) { root = new\_node; } else if (new\_node->data < parent->data) { parent->left = new\_node; } else { parent->right = new\_node; } ​ if (new\_node->parent == nullptr) { new\_node->color = "BLACK"; return; } ​ if (new\_node->parent->parent == nullptr) { return; } ​ fixInsert(new\_node); } ​ // Inorder traversal void inorder() { inorderHelper(root); } ​ // Search function Node\* search(int data) { return searchHelper(root, data); } }; ​ int main() { RedBlackTree rbt; ​ // Inserting elements rbt.insert(10); rbt.insert(20); rbt.insert(30); rbt.insert(15); ​ // Inorder traversal cout << "Inorder traversal:" << endl; rbt.inorder(); // Output: 10 15 20 30 ​ // Search for a node cout << "\nSearch for 15: " << (rbt.search(15) != rbt.search(0)) << endl; // Output: 1 (true) cout << "Search for 25: " << (rbt.search(25) != rbt.search(0)) << endl; // Output: 0 (false) ​ return 0; } Advantages of Red-Black Trees: Balanced: Red-Black Trees are self-balancing, meaning they automatically maintain a balance between the heights of the left and right subtrees. This ensures that search, insertion, and deletion operations take O(log n) time in the worst case. Efficient search, insertion, and deletion: Due to their balanced structure, Red-Black Trees offer efficient operations. Search, insertion, and deletion all take O(log n) time in the worst case. Simple to implement: The rules for maintaining the Red-Black Tree properties are relatively simple and straightforward to implement. Widely used: Red-Black Trees are a popular choice for implementing various data structures, such as maps, sets, and priority queues. Disadvantages of Red-Black Trees: More complex than other balanced trees: Compared to simpler balanced trees like AVL trees, Red-Black Trees have more complex insertion and deletion rules. Constant overhead: Maintaining the Red-Black Tree properties adds a small overhead to every insertion and deletion operation. Not optimal for all use cases: While efficient for most operations, Red-Black Trees might not be the best choice for applications where frequent insertions and deletions are required, as the constant overhead can become significant. Applications of Red-Black Trees: Implementing maps and sets: Red-Black Trees are often used to implement maps and sets, where efficient search, insertion, and deletion are crucial. Priority queues: Red-Black Trees can be used to implement priority queues, where elements are ordered based on their priority. File systems: Red-Black Trees are used in some file systems to manage file and directory structures. In-memory databases: Red-Black Trees are sometimes used in in-memory databases to store and retrieve data efficiently. Graphics and game development: Red-Black Trees can be used in graphics and game development for tasks like collision detection and pathfinding. Related Articles: Red-Black Tree definition & meaning in DSA Self-Balancing Binary Search Trees Red Black Tree vs AVL Tree What is the difference between Heap and Red-Black Tree? Insertion in Red-Black Tree Deletion in Red-Black Tree Red-Black Trees | Top-Down Insertion Comment More info Campus Training Program Next Article Red-Black Tree definition & meaning in DSA course-img 1417k+ interested Geeks Data Structures and Algorithms - Self Paced [Online Course] Explore course-img 112k+ interested Geeks Data Structures & Algorithms in JavaScript - Self Paced Course Explore course-img 200k+ interested Geeks C Programming Course Online - Learn C with Data Structures Explore geeksforgeeks-footer-logo Corporate & Communications Address: A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) Registered Address: K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305 GFG App on Play Store GFG App on App Store Advertise with us Company About Us Legal Privacy Policy Careers In Media Contact Us Corporate Solution Campus Training Program Explore Job-A-Thon Offline Classroom Program DSA in JAVA/C++ Master System Design Master CP Videos Tutorials Python Java C++ PHP GoLang SQL R Language Android DSA Data Structures Algorithms DSA for Beginners Basic DSA Problems DSA Roadmap DSA Interview Questions Competitive Programming Data Science & ML Data Science With Python Machine Learning ML Maths Data Visualisation Pandas NumPy NLP Deep Learning Web Technologies HTML CSS JavaScript TypeScript ReactJS NextJS NodeJs Bootstrap Tailwind CSS Python Tutorial Python Examples Django Tutorial Python Projects Python Tkinter Web Scraping OpenCV Tutorial Python Interview Question Computer Science GATE CS Notes Operating Systems Computer Network Database Management System Software Engineering Digital Logic Design Engineering Maths DevOps Git AWS Docker Kubernetes Azure GCP DevOps Roadmap System Design High Level Design Low Level Design UML Diagrams Interview Guide Design Patterns OOAD System Design Bootcamp Interview Questions School Subjects Mathematics Physics Chemistry Biology Social Science English Grammar Databases SQL MYSQL PostgreSQL PL/SQL MongoDB Preparation Corner Company-Wise Recruitment Process Aptitude Preparation Puzzles Company-Wise Preparation More Tutorials Software Development Software Testing Product Management Project Management Linux Excel All Cheat Sheets Courses IBM Certification Courses DSA and Placements Web Development Data Science Programming Languages DevOps & Cloud Programming Languages C Programming with Data Structures C++ Programming Course Java Programming Course Python Full Course Clouds/Devops DevOps Engineering AWS Solutions Architect Certification Salesforce Certified Administrator Course GATE 2026 GATE CS Rank Booster GATE DA Rank Booster GATE CS & IT Course - 2026 GATE DA Course 2026 GATE Rank Predictor @GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved Lightbox

**ChatGPT said:**

Thanks for sharing the detailed GeeksforGeeks article on **Red-Black Trees**!

Here's a quick summary and some key points from it, which might help you absorb the main concepts or use as notes:

**Red-Black Trees — Summary**

**What is a Red-Black Tree?**

* A self-balancing binary search tree.
* Each node has a color: **red** or **black**.
* Balances itself during insertions and deletions to maintain O(log n) height.

**Properties of Red-Black Trees**

1. Every node is either red or black.
2. The root is always black.
3. Red nodes cannot have red children (no two reds in a row).
4. Every path from a node to its descendant NIL nodes contains the same number of black nodes.
5. NIL leaves are considered black.

**Why Red-Black Trees?**

* Regular BSTs can become skewed (O(n) height).
* Red-Black Trees guarantee height ≤ 2 × log₂(n+1), ensuring operations run in O(log n).
* Operations like search, insert, delete are all O(log n).

**Comparison with AVL Trees**

* AVL trees are more strictly balanced, meaning faster searches but more rotations during insert/delete.
* Red-Black Trees are more flexible with balance, fewer rotations, better for frequent insertions/deletions.

**Basic Operations**

* **Insertion:** Insert like BST, then fix red-black violations using recoloring and rotations.
* **Deletion:** Remove node like BST, then fix double-black violations with rotations and recoloring.
* **Searching:** Same as BST search.
* **Rotations:** Left and Right rotations help rebalance the tree.

**Rotations Explained**

* **Left rotation:** Node moves down left, right child moves up.
* **Right rotation:** Node moves down right, left child moves up.

**Advantages**

* Self-balancing with guaranteed O(log n) height.
* Efficient insert/search/delete.
* Used in many libraries (e.g., C++ STL's map and set).

**Disadvantages**

* More complex insertion/deletion logic compared to AVL.
* Slight overhead on each operation due to balancing.

**Applications**

* Implementing maps and sets.
* Priority queues.
* File systems.
* In-memory databases.
* Graphics and game development.

If you'd like, I can help you with:

* Explaining any part in detail.
* Implementing specific operations in code (C++/Java/Python).
* Visualizing insertions, deletions, rotations step-by-step.
* Comparing Red-Black Trees with AVL or other balanced trees.
* Practice problems on Red-Black Trees.

Just let me know what you want!

**You said:**

Segment Tree about this

**ChatGPT said:**

Got it! You want to know about **Segment Trees**, another important data structure often used in algorithms and competitive programming, especially for range queries and updates.

**Segment Tree Overview**

**What is a Segment Tree?**

A **Segment Tree** is a **binary tree data structure** that allows answering range queries and updates efficiently on an array. It is useful for problems where you need to perform operations (like sum, min, max, gcd, etc.) on subarrays repeatedly.

**Why use Segment Tree?**

* Naive approach for range queries (like sum from index L to R) is O(n).
* With Segment Tree, you can do both:
  + **Query in O(log n)**
  + **Update in O(log n)**

This makes it efficient for multiple queries and updates on an array.

**Basic Idea**

* The array is represented as a tree, where:
  + The root represents the whole array.
  + Each internal node represents a segment (subarray) of the original array.
  + Leaf nodes represent single elements.
* Each node stores the aggregate information for its segment (sum, min, max, etc.).

**Structure**

For an array of size n, the segment tree size is about 4\*n nodes to store all segments.

**Operations**

1. **Build Tree**
   * Recursively build the tree by dividing the array into two halves.
2. **Query (Range Query)**
   * Given a range [L, R], find the aggregate (sum/min/max/etc.) over that range.
3. **Update**
   * Update a specific element or a range of elements, and update the tree accordingly.

**Example: Sum Segment Tree**

**Array:** [1, 3, 5, 7, 9, 11]

* Root stores sum of entire array.
* Left child stores sum of left half.
* Right child stores sum of right half.
* Leaf nodes store individual elements.

**Time Complexity**

* Building the tree: **O(n)**
* Query: **O(log n)**
* Update: **O(log n)**

**Simple C++ Code for Sum Segment Tree**

#include <iostream> #include <vector> using namespace std; class SegmentTree { vector<int> tree; int n; void build(vector<int>& arr, int idx, int start, int end) { if (start == end) { tree[idx] = arr[start]; return; } int mid = (start + end) / 2; build(arr, 2 \* idx + 1, start, mid); build(arr, 2 \* idx + 2, mid + 1, end); tree[idx] = tree[2 \* idx + 1] + tree[2 \* idx + 2]; } int query(int idx, int start, int end, int l, int r) { *// Completely outside range* if (r < start || l > end) return 0; *// Completely inside range* if (l <= start && end <= r) return tree[idx]; *// Partially inside and outside* int mid = (start + end) / 2; return query(2 \* idx + 1, start, mid, l, r) + query(2 \* idx + 2, mid + 1, end, l, r); } void update(int idx, int start, int end, int pos, int val) { if (start == end) { tree[idx] = val; return; } int mid = (start + end) / 2; if (pos <= mid) update(2 \* idx + 1, start, mid, pos, val); else update(2 \* idx + 2, mid + 1, end, pos, val); tree[idx] = tree[2 \* idx + 1] + tree[2 \* idx + 2]; } public: SegmentTree(vector<int>& arr) { n = arr.size(); tree.resize(4 \* n); build(arr, 0, 0, n - 1); } int query(int l, int r) { return query(0, 0, n - 1, l, r); } void update(int pos, int val) { update(0, 0, n - 1, pos, val); } }; int main() { vector<int> arr = {1, 3, 5, 7, 9, 11}; SegmentTree seg(arr); cout << "Sum of values in range [1, 3]: " << seg.query(1, 3) << endl; *// Output: 3+5+7=15* seg.update(1, 10); *// Update index 1 to value 10* cout << "Sum of values in range [1, 3] after update: " << seg.query(1, 3) << endl; *// Output: 10+5+7=22* return 0; }

**Key Points:**

* Leaf nodes store individual elements.
* Internal nodes store aggregated values of their children.
* Queries and updates affect only O(log n) nodes due to tree structure.

If you want, I can:

* Explain the code in detail.
* Show other variations (like range min query, max query).
* Explain lazy propagation for range updates.
* Provide problems to practice Segment Trees.

Would you like that?

Top of Form

Bottom of Form

ChatGPT can make mistakes. Check important info. See Cookie Preferences.