💯

Interview Questions

1. What is an Array? Explain its time and space complexities.

- **Array**: A collection of elements stored at contiguous memory locations.
- **Access time**: O(1) – Direct access using index.
- **Search time**: O(n) – Linear search for an element.
- **Insertion/Deletion time**:
  - O(n) for inserting/deleting at a random position.
  - O(1) for adding at the end (if space is available).
- **Space Complexity**: O(n) for n elements.

Example use case: Storing the **scores** of players in a game.

---

2. What is a Linked List, and how is it different from an Array?

- **Linked List**: A linear data structure where each element points to the next.
- **Time Complexity**:
  - Access: O(n) – No direct access by index.

    - Insertion/Deletion: O(1) if done at the head or tail, O(n) if at an arbitrary position.
- **Space Complexity**: O(n) for n nodes.
- **Differences from Array**:
    - Dynamic size, no contiguous memory allocation.
    - Slower access but easier insertion/deletion.

Example use case: **Undo functionality** in applications.

---

3. What is a Stack?

- **Stack**: A LIFO (Last In, First Out) data structure.
- Operations:
    - Push: O(1)
    - Pop: O(1)
    - Peek: O(1)
- **Space Complexity**: O(n).

Example use case: **Browser history** management.

---

4. What is a Queue? How is it different from a Stack?

- **Queue**: A FIFO (First In, First Out) data structure.
- Operations:
    - Enqueue: O(1)
    - Dequeue: O(1)
- **Difference from Stack**: Queue follows **FIFO**, Stack follows **LIFO**.

Example use case: **Task scheduling** in operating systems.

---

5. Explain Recursion with a real-world example.

- **Recursion**: A function that calls itself until a base condition is met.
- **Time Complexity**: Depends on the problem. Example – Factorial: O(n).
- Example: **Towers of Hanoi** problem.

---

6. What is Backtracking? Provide an example problem.

- **Backtracking**: A technique to explore all possible solutions by exploring each path step by step.
- Example: **N-Queens problem** – Placing N queens on an N×N chessboard.
- **Time Complexity**: O(n!), **Space Complexity**: O(n).

---

7. Explain Binary Search and its time complexity.

- **Binary Search**: A search algorithm that works on sorted arrays.
- **Time Complexity**: O(log n) for best, average, and worst cases.
- **Space Complexity**: O(1).

---

8. What is Merge Sort? Explain with time and space complexities.

- **Merge Sort**: A divide-and-conquer sorting algorithm.
- **Time Complexity**: O(n log n) for all cases.
- **Space Complexity**: O(n) due to temporary arrays.
- **Stable Sort**: Preserves the relative order of elements.

---

9. What is Quick Sort? Why is it preferred over Merge Sort?

- **Quick Sort**: A divide-and-conquer algorithm using a pivot.
- **Time Complexity**:
  - Best/Average: O(n log n).
  - Worst: $O(n^2)$ (when pivot is poorly chosen).
- **Space Complexity**: O(log n).
- Preferred because it **uses less space** than Merge Sort.

---

10. Explain Min-Heap and Max-Heap with time complexities.

- **Min-Heap**: Root node contains the smallest element.
- **Max-Heap**: Root node contains the largest element.
- Operations:

- Insertion/Deletion: O(log n).
- Accessing Min/Max: O(1).
- **Space Complexity**: O(n).

Example use case: **Priority queues**.

---

11. What is a HashMap? How does it work?

- **HashMap**: A key-value data structure that uses **hashing** for fast lookups.
- **Time Complexity**:
  - Best case: O(1).
  - Worst case: O(n) (in case of hash collisions).
- **Space Complexity**: O(n).

Example: **Storing user data** with usernames as keys.

---

12. What is a Priority Queue? How is it implemented?

- **Priority Queue**: A data structure where elements are accessed based on priority.
- Typically implemented using **heaps**.
- **Time Complexity**:
  - Insertion: O(log n).
  - Accessing highest priority: O(1).

---

13. Explain Time Complexity and Space Complexity. Why are they important?

- **Time Complexity**: Measures the **time taken** by an algorithm as input size grows.
- **Space Complexity**: Measures the **amount of memory** used.
- Important to ensure the program is **efficient and scalable**.

---

14. What is a Graph? What are its types?

- **Graph**: A collection of nodes (vertices) connected by edges.

- **Types**:
  - Directed and Undirected.
  - Weighted and Unweighted.
- Example: **Social networks** are represented as graphs.

---

15. Explain DFS and BFS.

- **DFS (Depth-First Search)**: Explores as far as possible along a branch.
- **BFS (Breadth-First Search)**: Explores all neighbors first.

---

16. What are the different types of Trees?

- **Binary Tree**: Each node has at most 2 children.
- **Binary Search Tree (BST)**: Left child < Parent < Right child.
- **AVL Tree**: Self-balancing BST.

---

17. What is a Hash Collision? How can it be resolved?

- **Hash Collision**: When two keys produce the same hash value.
- **Resolution Techniques**:
  - **Chaining**: Store multiple elements in the same bucket.
  - **Open Addressing**: Use linear probing to find the next available slot.

---

18. What is Dynamic Programming? Provide an example.

- **Dynamic Programming**: Solves problems by breaking them into overlapping subproblems.
- Example: **Fibonacci sequence** with memoization.
- **Time Complexity**: O(n).

---

19. What is the difference between Greedy and Dynamic Programming?

- **Greedy**: Makes local optimal choices at each step.

- **Dynamic Programming**: Considers all possible solutions to find the global optimum.

---

20. Explain the difference between Linear Search and Binary Search.

- **Linear Search**: O(n) time, works on unsorted data.
- **Binary Search**: O(log n) time, works only on sorted data.

---

21. What is Selection Sort? Provide its time and space complexities.

- **Selection Sort**: Repeatedly selects the smallest element from the unsorted portion and swaps it with the first unsorted element.
- **Time Complexity**:
  - Best: $O(n^2)$
  - Average: $O(n^2)$
  - Worst: $O(n^2)$
- **Space Complexity**: O(1) (In-place sorting).

Example use case: **Sorting a small dataset** where simplicity is preferred over speed.

---

22. What is Insertion Sort? How does it work?

- **Insertion Sort**: Builds the sorted array one element at a time by comparing it with previous elements.
- **Time Complexity**:
  - Best: O(n) (already sorted)
  - Average/Worst: $O(n^2)$
- **Space Complexity**: O(1).

Example: **Sorting playing cards** by hand.

---

23. What is Radix Sort? When is it useful?

- **Radix Sort**: Non-comparative sorting algorithm that sorts numbers digit by digit.
- **Time Complexity**: O(d * (n + k)) where d = number of digits and k = range of digits.
- **Space Complexity**: O(n + k).
- Useful for: Sorting **large integers or strings** efficiently.

---

24. Explain Heap Sort with its complexities.

- **Heap Sort**: A comparison-based sorting technique that uses a **heap** to sort elements.
- **Time Complexity**:
  - Best/Average/Worst: O(n log n).
- **Space Complexity**: O(1) (in-place sorting).
- Application: Used in **priority queue** implementations.

---

25. What is the difference between Linear Probing and Chaining in Hashing?

- **Linear Probing**: In case of a collision, the next empty slot is used.
  - **Time Complexity**: O(1) in best case, O(n) in worst case.
- **Chaining**: Each bucket holds a **linked list** of entries.
  - **Time Complexity**: O(1) on average, O(n) in worst case.

---

26. What are AVL Trees? How do they maintain balance?

- **AVL Tree**: A self-balancing Binary Search Tree (BST).
- **Balancing Condition**: The **height difference** (balance factor) between left and right subtrees must be at most 1.
- **Time Complexity**: O(log n) for insertion, deletion, and search.

Application: **Databases** for fast lookups.

---

27. What is the difference between BFS and DFS in Graphs?

- **BFS (Breadth-First Search)**: Explores neighbors level by level.

    - **Time Complexity**: O(V + E)
- **DFS (Depth-First Search)**: Explores deep into one branch before backtracking.
    - **Time Complexity**: O(V + E)
- **Applications**:
    - BFS: **Shortest path** algorithms.
    - DFS: **Cycle detection** and **topological sorting**.

---

28. How does Dijkstra's Algorithm work? What are its limitations?

- **Dijkstra's Algorithm**: Finds the shortest path from a source node to all other nodes in a graph with **non-negative weights**.
- **Time Complexity**: O((V + E) log V) using a priority queue.
- Limitation: Does not work with **negative edge weights**.

---

29. What is Bellman-Ford Algorithm? How is it different from Dijkstra's?

- **Bellman-Ford Algorithm**: Finds shortest paths and can handle **negative edge weights**.
- **Time Complexity**: O(V * E).
- **Difference**: Bellman-Ford is slower but more versatile than Dijkstra's.

---

30. What is a Trie? Where is it used?

- **Trie**: A tree-based data structure used for storing a dynamic set of strings.
- **Time Complexity**: O(n) for search, where n is the length of the word.
- Application: **Autocomplete** and **spell-checking**.

---

31. What is Dynamic Memory Dispatch in Java?

- **Dynamic Memory Dispatch**: Method overriding in Java where the call to an overridden method is resolved at **runtime**.
- Example:

```
class Animal {
    void sound() { System.out.println("Animal makes a so
und"); }
}
class Dog extends Animal {
    void sound() { System.out.println("Dog barks"); }
}
Animal a = new Dog();
a.sound(); // Output: Dog barks
```

32. What are Red-Black Trees?

- **Red-Black Tree**: A self-balancing binary search tree where each node is either **red** or **black**.
- Properties:
  - Root is always black.
  - No two red nodes can be adjacent.
- **Time Complexity**: O(log n) for insertion, deletion, and search.

33. How does a HashMap handle collisions?

- HashMap resolves collisions using **chaining** or **open addressing**.
- In Java, it uses **linked lists** for chaining and converts them to **balanced trees** if collisions become excessive.

34. What is a Segment Tree? Where is it used?

- **Segment Tree**: A tree data structure used for **range queries**.
- **Time Complexity**: O(log n) for queries and updates.
- Application: **Range Sum Queries**.

35. Explain Time Complexities of Binary Search Tree (BST) operations.

1. **Search**:
   - **Best Case**: O(1) (if the root node is the target).
   - **Average Case**: O(log n) (when the tree is balanced).
   - **Worst Case**: O(n) (in case of a skewed tree).
2. **Insertion**:
   - **Best Case**: O(1) (if inserting into an empty tree).
   - **Average Case**: O(log n) (when the tree is balanced).
   - **Worst Case**: O(n) (if the tree is skewed).
3. **Deletion**:
   - **Best Case**: O(1) (deleting a leaf node).
   - **Average Case**: O(log n) (when the tree is balanced).
   - **Worst Case**: O(n) (if the tree is skewed).

Space Complexity

- The space complexity of a BST is O(n) for storing n nodes, regardless of its shape.

---

36. What is KMP Algorithm? How is it better than Naïve String Matching?

- **KMP (Knuth-Morris-Pratt)**: String matching algorithm using **prefix tables** to avoid redundant comparisons.
- **Time Complexity**: O(n + m), where n = text length and m = pattern length.

---

37. Explain Greedy Algorithms. When are they useful?

- **Greedy Algorithms**: Make the best local choice at each step.
- Application: **Huffman Encoding**, **Kruskal's Algorithm**.

---

38. What is Ternary Search? How does it compare to Binary Search?

- **Ternary Search**: Similar to binary search but splits the search space into **three parts**.

- **Time Complexity**: $O(\log_3 n)$.
- Less efficient than Binary Search in practice due to **extra comparisons**.

---

39. What is Floyd-Warshall Algorithm? What is its time complexity?

- **Floyd-Warshall Algorithm**: Finds shortest paths between all pairs of vertices in a graph.
- **Time Complexity**: $O(V^3)$.

---

40. What are Sliding Window Algorithms?

- **Sliding Window**: Optimizes problems involving contiguous subarrays or sequences.
- Example: Find the **maximum sum subarray** of size k.
- Time Complexity: O(n).

---

41. Explain the concept of Memoization in Dynamic Programming.

- **Memoization**: Storing results of subproblems to avoid recomputation.
- Example: Fibonacci series calculation with **memoization** reduces time complexity to O(n).

---

42. What is LRU Cache? How is it implemented?

- **LRU Cache (Least Recently Used)**: Caches the most recently used elements and removes the least recently accessed ones.
- **Time Complexity**: O(1) for access using a **HashMap + Doubly Linked List**.

---

43. What is Kruskal's Algorithm? How does it find Minimum Spanning Tree?

- **Kruskal's Algorithm**: Greedily selects the smallest edge that does not form a cycle.
- **Time Complexity**: O(E log E), where E is the number of edges.

---

44. Explain the difference between Min-Heap and Max-Heap.

- **Min-Heap**: Root contains the smallest element.
- **Max-Heap**: Root contains the largest element.
- **Time Complexity**: O(log n) for insertion and deletion.

---

45. What is the difference between a Subset and a Subsequence?

- **Subset**: Any collection of elements from a set, including an **empty set**.
    - Example: Subsets of `{1, 2}` → `{}`, `{1}`, `{2}`, `{1, 2}`.
- **Subsequence**: A sequence derived by deleting elements from the original array **without changing the order**.
    - Example: Subsequences of `[1, 2, 3]` → `[1]`, `[2, 3]`, `[1, 3]`.
- **Total Subsets**: 2^n for a set with n elements.
- **Applications**: Subsets are used in **combinatorics**, while subsequences are important in **string matching algorithms**.

---

46. What is the difference between Linear Search and Binary Search?

- **Linear Search**: Checks elements one by one; works on **unsorted** data.
    - **Time Complexity**: O(n)
    - **Use case**: Small datasets or unsorted lists.
- **Binary Search**: Divides the sorted array into halves; requires **sorted input**.
    - **Time Complexity**: O(log n)
    - **Application**: Efficient **searching** in sorted data, e.g., finding a number in a phonebook.

---

47. Complexity of a program to reverse a string using recursion.

- **Time Complexity**: O(n)
- **Space Complexity**: O(n) due to recursive stack.

- **Application**: Useful in **text processing** or **palindrome checks**.

---

48. How does a Two-Pointer approach work? Provide an example.

- **Two-pointer technique**: Uses two pointers to solve **searching or partitioning problems** efficiently.
  - Example: Finding if a sorted array has two elements with a given sum.

```
boolean hasPairWithSum(int[] arr, int sum) {
    int left = 0, right = arr.length - 1;
    while (left < right) {
        int currentSum = arr[left] + arr[right];
        if (currentSum == sum) return true;
        else if (currentSum < sum) left++;
        else right--;
    }
    return false;
}
```

- **Time Complexity**: O(n)
- **Application**: Solving **array partitioning** problems.

---

49. What is Backtracking? Provide a real-world example.

- **Backtracking**: A problem-solving technique where we **explore all possibilities** and backtrack when a solution fails.
  - Example: **N-Queens problem** or **solving Sudoku**.
- **Time Complexity**: Exponential in nature, O(2^n) for generating all subsets.
- **Application**: **Puzzle solving**, **pathfinding** in a maze, etc.

---

50. Explain the Sliding Window Technique with an example.

- **Sliding Window**: A technique used for **subarray or substring problems** to reduce time complexity.
  - Example: Find the **maximum sum subarray** of size k.

```
java
Copy code
int maxSum(int[] arr, int k) {
    int maxSum = 0, windowSum = 0;
    for (int i = 0; i < k; i++) windowSum += arr[i];
    maxSum = windowSum;
    for (int i = k; i < arr.length; i++) {
        windowSum += arr[i] - arr[i - k];
        maxSum = Math.max(maxSum, windowSum);
    }
    return maxSum;
}
```

- **Time Complexity**: O(n)
- **Application**: **Finding max/min subarrays** or **longest substrings**.

---

51. What is Kadane's Algorithm? How is it used?

- **Kadane's Algorithm**: Finds the **maximum sum subarray** in O(n) time.
- **Time Complexity**: O(n)
- Example:

```
java
Copy code
int maxSubArraySum(int[] arr) {
    int maxSoFar = arr[0], maxEndingHere = arr[0];
    for (int i = 1; i < arr.length; i++) {
        maxEndingHere = Math.max(arr[i], maxEndingHere +
arr[i]);
        maxSoFar = Math.max(maxSoFar, maxEndingHere);
    }
    return maxSoFar;
}
```

---

52. What is the Longest Common Subsequence (LCS)? How is it computed?

- **LCS**: Finds the **longest subsequence common** between two sequences.
- **Time Complexity**: O(m * n) for two strings of lengths m and n.
- **Application**: Used in **DNA sequence matching** and **text comparison**.

---

53. Explain the difference between Stack and Queue.

- **Stack**: LIFO (Last In, First Out) structure.
  - Example: **Function call stack**.
- **Queue**: FIFO (First In, First Out) structure.
  - Example: **Task scheduling** in an OS.

---

54. How does a Priority Queue work? What is its use case?

- **Priority Queue**: Each element has a priority, and **higher priority elements** are dequeued first.
  - Example: **Dijkstra's algorithm**.
- **Time Complexity**: O(log n) for insertion and deletion using a **min-heap**.

---

55. What is the difference between Recursion and Iteration?

- **Recursion**: A function calls itself to solve a problem.
  - Example: **Factorial** using recursion.
- **Iteration**: Uses loops to repeat a task.
  - Example: Calculating factorial with a **for-loop**.
- **Recursion uses more stack space**; iteration is usually more efficient.

---

56. Explain Hashing. What are its real-world applications?

- **Hashing**: Maps data to a fixed size using **hash functions**.
- **Applications**:
  - **HashMap** in Java.
  - **Password storage** in databases.

57. How is a Singly Linked List different from a Doubly Linked List?

- **Singly Linked List**: Each node has only one pointer to the next node.
- **Doubly Linked List**: Each node has **two pointers**, one to the next and one to the previous node.
- **Application**:
  - Singly: **Stack implementation**.
  - Doubly: **Navigation systems**.

---

58. What is the purpose of Depth-First Search (DFS) and Breadth-First Search (BFS)?

- **DFS**: Explores as deep as possible into the graph before backtracking.
  - **Use case**: **Topological sorting**.
- **BFS**: Explores level by level.
  - **Use case**: **Shortest path** in unweighted graphs.