TOP DSA THEORY QUESTIONS

Praful Sharma

KN ACADEMY | EDUCATIONAL INSTITUTE

Contents

Time Complexity

Q1. What is time complexity, and why is it important?

✅ **Solution:**
Time complexity describes the **efficiency** of an algorithm by measuring how the runtime increases with input size (**n**). It helps compare different algorithms and choose the most efficient one.

Q2. What is Big-O notation, and how is it used?

✅ **Solution:**
Big-O notation expresses the **upper bound** of an algorithm's time complexity.

- Example: A loop running **n** times has **O(n)** complexity.
- A nested loop running n^2 times has **O(n^2)** complexity.

Q3. Explain best-case, worst-case, and average-case complexity with an example.

✅ **Solution:**
For **Linear Search (O(n))** on an array of size **n**:

- **Best-case**: Element found at the first index → **O(1)**
- **Worst-case**: Element not present or last element → **O(n)**

- **Average-case**: **O(n/2) = O(n)** (since constants are ignored in Big-O).

Q4. What is O(1) time complexity? Give an example.

✅ **Solution:**
O(1) means **constant time**, independent of input size.
Example:

```
def get_first_element(arr):
    return arr[0]  # Always takes constant time
```

This function runs in **O(1)** time.

Q5. What is O(log n) time complexity? Give an example.

✅ **Solution:**
O(log n) means the time **reduces exponentially** as input grows.
Example: **Binary Search** (halving input each step).

---

2. Sorting Algorithms

Q6. What is the time complexity of Bubble Sort?

✅ **Solution:**

- **Worst-case:** $O(n^2)$ (if array is in reverse order).
- **Best-case:** O(n) (if array is already sorted).

Q7. What is the time complexity of Quick Sort?

✅ **Solution:**

- **Best/Average-case:** O(n log n)
- **Worst-case:** $O(n^2)$ (if pivot selection is poor).

Q8. Why is Merge Sort O(n log n)?

✅ **Solution:**
Merge Sort **divides** the array into halves (**log n**) and **merges** them in O(n).
Formula: **T(n) = 2T(n/2) + O(n)** → Solves to **O(n log n)**.

Q9. Why is Counting Sort faster than Quick Sort?

✅ **Solution:**
Counting Sort runs in **O(n + k)** (where k is the range of elements) and is **non-comparative**, unlike Quick Sort which is **O(n log n)** in most cases.

Q10. What is the best sorting algorithm in terms of time complexity?

✅ **Solution:**

- If **comparison-based**: **Merge Sort / Quick Sort (O(n log n))**.
- If **non-comparison-based**: **Counting Sort / Radix Sort (O(n))**.

---

3. Searching Algorithms

Q11. What is the time complexity of Linear Search?

✅ **Solution:**

- **Best-case:** O(1) (first element match).
- **Worst-case:** O(n) (element not present).

Q12. What is the time complexity of Binary Search?

✅ **Solution:**
Binary Search halves the input each step → **O(log n)**.

Q13. Why is Jump Search better than Linear Search?

✅ **Solution:**
Jump Search checks at fixed intervals of **√n**, making it **O(√n)**, faster than **O(n)** for large arrays.

Q14. What is the time complexity of Hash Table search?

✅ **Solution:**

- **Best/Average-case:** O(1)
- **Worst-case:** O(n) (in case of hash collisions).

Q15. What is the time complexity of Fibonacci Search?

✅ **Solution:**

- **O(log n)** (similar to Binary Search).

---

4. Time Complexity in Recursion

Q16. Solve the recurrence relation T(n) = 2T(n/2) + O(n).

✅ **Solution:**
Using **Master's Theorem**,

- a = 2, b = 2, f(n) = O(n) → **O(n log n)**.

Q17. What is the time complexity of recursive Fibonacci?

✅ **Solution:**
T(n) = T(n-1) + T(n-2) → **$O(2^n)$** (exponential).

Q18. What is the time complexity of Tower of Hanoi?

✅ **Solution:**
T(n) = 2T(n-1) + O(1) → **$O(2^n)$**.

Q19. How can we optimize recursive Fibonacci?

✅ **Solution:**
Use **Memoization (O(n))** or **Dynamic Programming (O(n))**.

Q20. What is the time complexity of Merge Sort using recursion?

✅ **Solution:**
T(n) = 2T(n/2) + O(n) → **O(n log n)**.

---

5. Advanced Complexity Analysis

Q21. What is Amortized Time Complexity?

✅ **Solution:**
Analyzing the **average** cost per operation over a sequence of operations.
Example: **Dynamic array resizing (O(1) amortized insertions)**.

Q22. What is the time complexity of Dijkstra's Algorithm?

✅ **Solution:**

- **Using Min Heap:** O((V + E) log V).
- **Using Simple Array:** $O(V^2)$.

Q23. What is the time complexity of Floyd-Warshall Algorithm?

✅ **Solution:**
$O(V^3)$ (for all-pairs shortest path).

Q24. What is the time complexity of Kruskal's Algorithm?

✅ **Solution:**
O(E log E) (for Minimum Spanning Tree).

Q25. What is the time complexity of Bellman-Ford Algorithm?

✅ **Solution:**
O(VE) (for shortest path in weighted graphs).

---

6. Miscellaneous

Q26. What is the time complexity of matrix multiplication?

✅ **Solution:**

- **Naive:** $O(n^3)$.
- **Strassen's Algorithm:** $O(n^{2.80})$.

Q27. What is the time complexity of checking if a number is prime?

✅ **Solution:**

- **Brute Force:** $O(\sqrt{n})$.
- **Optimized (Sieve of Eratosthenes):** O(n log log n).

Q28. What is the time complexity of finding GCD using Euclidean Algorithm?

✅ **Solution:**
O(log n).

Q29. What is the time complexity of finding LCM?

✅ **Solution:**
O(log n) (using GCD-based formula).

Q30. What is the time complexity of factorial calculation?

✅ **Solution:**

- **Recursive:** O(n).
- **Iterative:** O(n).

Basic Arrays Theory

1. What is an array in programming?

Solution: An array is a data structure that stores a fixed-size sequence of elements of the same data type. It allows random access using an index.

2. How are arrays stored in memory?

Solution: Arrays are stored in contiguous memory locations. The address of each element is calculated as:

Address of A[i] = Base Address + (i * Size of data type)

3. What are the advantages of using arrays?

Solution:

- Fast access using index (O(1) time complexity)
- Efficient memory utilization
- Easy to traverse and sort

4. What are the disadvantages of arrays?

Solution:

- Fixed size (cannot dynamically resize)
- Insertion and deletion operations are costly (O(n) in the worst case)

5. What is the difference between an array and a linked list?

Solution:

- **Array:** Uses contiguous memory, provides O(1) access but insertion/deletion is costly.
- **Linked List:** Uses non-contiguous memory, insertion/deletion is easy but access is O(n).

6. How do you declare and initialize an array in C++ and Java?

Solution:

- **C++:**

```
int arr[5] = {1, 2, 3, 4, 5};
```

- **Java:**

int[] arr = {1, 2, 3, 4, 5};

7. What is a dynamic array, and how is it different from a static array?

Solution:

- A **static array** has a fixed size, whereas a **dynamic array** (like ArrayList in Java) can grow or shrink dynamically.
- Example in C++ using vector:

```
vector<int> arr;
arr.push_back(10); // Adds element dynamically
```

Intermediate Array Questions:

8. What is the difference between row-major and column-major order in a 2D array?

Solution:

- **Row-major order:** Elements are stored row-wise (default in C/C++).
- **Column-major order:** Elements are stored column-wise (default in Fortran).

9. How does array slicing work in Python?

Solution:

- Slicing allows accessing parts of an array:

```
arr = [10, 20, 30, 40, 50]
print(arr[1:4])  # Output: [20, 30, 40]
```

10. How do you reverse an array in-place?

Solution: Using two-pointer technique:

```
void reverseArray(int arr[], int n) {
   int left = 0, right = n - 1;
   while (left < right) {
      swap(arr[left], arr[right]);
      left++;
      right--;
   }
}
```

11. How do you find duplicate elements in an array?

Solution: Using hashing:

```
unordered_map<int, int> freq;
for (int i = 0; i < n; i++) {
    if (freq[arr[i]]++) cout << arr[i] << " is duplicate\n";
}
```

Arrays

1. What is an array?

Answer:
An array is a **fixed-size** data structure that stores elements of the same data type in **contiguous memory locations**. It allows easy indexing and retrieval of elements.

Example (C++):

```
int arr[5] = {1, 2, 3, 4, 5};  // Integer array of size 5
```

---

2. How is an array different from a linked list?

Feature	Array	Linked List
Memory Location	Contiguous	Non-contiguous
Size	Fixed	Dynamic
Insertion/Deletion	Costly (O(n))	Efficient (O(1) or O(n))
Access Time	O(1) (Direct Indexing)	O(n) (Traversal required)

---

3. What are the advantages and disadvantages of arrays?

Advantages:

- Fast access time **O(1)** due to direct indexing.
- Better cache locality.

Disadvantages:

- Fixed size (cannot grow dynamically).
- Insertion and deletion operations are expensive **O(n)**.

---

4. How do you declare and initialize an array?

Example (Java):

int[] arr = {10, 20, 30, 40}; // Declaration & Initialization

Example (Python):

arr = [10, 20, 30, 40] # List works like an array in Python

---

5. What is the difference between a one-dimensional and a multi-dimensional array?

- **One-Dimensional Array**: int arr[5] = {1, 2, 3, 4, 5};
- **Multi-Dimensional Array**:

int arr[2][3] = { {1, 2, 3}, {4, 5, 6} };

---

6. How do you access elements in an array?

Example (C++):

int arr[] = {10, 20, 30, 40};

cout << arr[2]; // Output: 30

---

7. What is an "array index out of bounds" error?

Answer:
Occurs when accessing an index outside the array size.
Example:

int arr[5] = {1, 2, 3, 4, 5};

cout << arr[10]; // Undefined behavior

Intermediate Questions:

8. What is the time complexity for accessing an element in an array?

- **O(1) – Constant time** (since arrays allow direct indexing).

---

9. How can you insert or delete an element in an array?

Insertion (At Index pos):

```
void insertAt(int arr[], int &n, int pos, int value) {
   for (int i = n; i > pos; i--)
      arr[i] = arr[i-1]; // Shift elements
```

```
    arr[pos] = value;
    n++;
}
```

Deletion (At Index pos):

```
void deleteAt(int arr[], int &n, int pos) {
    for (int i = pos; i < n-1; i++)
        arr[i] = arr[i+1]; // Shift elements
    n--;
}
```

Time Complexity: O(n) (because of shifting).

---

10. How do you find the largest element in an array?

Solution (C++):

```
int findLargest(int arr[], int n) {
    int maxVal = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > maxVal) maxVal = arr[i];
    return maxVal;
}
```


Time Complexity: O(n)

Advanced Questions:

11. How do you reverse an array?

Solution (Python):

```
arr = [1, 2, 3, 4, 5]
arr.reverse()  # In-place reversal
print(arr)  # Output: [5, 4, 3, 2, 1]
```

Time Complexity: O(n)

---

12. How do you find duplicate elements in an array?

Solution (C++ using HashMap):

```
#include <iostream>
#include <unordered_map>
using namespace std;

void findDuplicates(int arr[], int n) {
    unordered_map<int, int> freq;
    for (int i = 0; i < n; i++) freq[arr[i]]++;
    for (auto x : freq)
        if (x.second > 1) cout << x.first << " ";
}

int main() {
    int arr[] = {1, 2, 3, 4, 2, 1, 5};
    findDuplicates(arr, 7); // Output: 1 2
}
```


Time Complexity: O(n)

---

13. How do you sort an array?

Solution (C++ using STL Sort):

```
#include <algorithm>
sort(arr, arr+n);
```

Time Complexity: O(n log n)

---

14. How do you find the missing number in an array of size N containing numbers from 1 to N?

Mathematical Approach:

```
int missingNumber(int arr[], int n) {
    int sum = (n * (n + 1)) / 2;
    for (int i = 0; i < n - 1; i++)
```

```
        sum -= arr[i];
    return sum;
}
```

Time Complexity: O(n)

---

15. How do you check if an array is sorted?

Solution (C++):

```
bool isSorted(int arr[], int n) {
    for (int i = 1; i < n; i++)
        if (arr[i] < arr[i - 1]) return false;
    return true;
}
```

Searching Algorithms

Q1. What is a searching algorithm? Why is it important?

✅ **Solution:**
A **searching algorithm** is a method for **finding an element** in a data structure (array, list, tree, etc.). It is crucial for **retrieving, modifying, or deleting data efficiently**.

Q2. What are the two main types of searching algorithms?

✅ **Solution:**

1. **Linear Search (Sequential Search)** – O(n)
2. **Binary Search (Divide & Conquer)** – O(log n)

Q3. What is the difference between linear and binary search?

✅ **Solution:**

Feature	**Linear Search**	**Binary Search**
Data Requirement	Works on unsorted data	Works only on sorted data

Feature	Linear Search	Binary Search
Time Complexity	O(n)	O(log n)
Space Complexity	O(1)	O(1)
Best-case Complexity	O(1) (first element match)	O(1) (middle element match)
Worst-case Complexity	O(n)	O(log n)

Q4. When should you use Linear Search over Binary Search?

✅ **Solution:**

- When the dataset is **small**.
- When the data is **unsorted** and **one-time search** is needed (sorting costs extra time).
- When the dataset is **dynamic** (frequent insertions/deletions).

Q5. What is the worst-case time complexity of Linear Search?

✅ **Solution:**

- **O(n)** (when the element is at the last index or not present).

Q6. What is the best-case time complexity of Binary Search?

✅ **Solution:**

- **O(1)** (if the element is at the middle index).

Q7. Why is Binary Search better than Linear Search for large datasets?

✅ **Solution:**

- Binary Search **divides** the search space into **halves** at each step, making it **O(log n)**.
- Linear Search **checks each element** one by one, making it **O(n)**.
- **Example**: Searching in **1 million** elements:
  - **Linear Search**: ~1,000,000 comparisons
  - **Binary Search**: ~$\log_2(1{,}000{,}000) \approx 20$ comparisons (much faster!)

Q8. What is the difference between Recursive and Iterative Binary Search?

✅ **Solution:**

Feature	Recursive Binary Search	Iterative Binary Search
Implementation	Uses recursion	Uses loops
Space Complexity	O(log n) (due to recursive calls)	O(1)
Performance	Slightly slower due to function call overhead	Faster due to no recursion overhead
Use Case	Suitable for problems with recursive nature	More efficient for large inputs

---

3. Advanced Searching Algorithms

Q9. What is Jump Search? When is it useful?

✅ **Solution:**
Jump Search is an **improvement over Linear Search** for **sorted** arrays.

- Instead of checking elements **one by one**, it jumps **√n** steps at a time.
- **Best-case Complexity:** O(1)
- **Worst-case Complexity:** O(√n)

Example Implementation:

```
def jump_search(arr, key):
    n = len(arr)
    step = int(math.sqrt(n))
    prev, curr = 0, step
    while curr < n and arr[curr] < key:
        prev = curr
        curr += step
    for i in range(prev, min(curr, n)):
        if arr[i] == key:
            return i
    return -1
```


Use Case: When data is **sorted and large** but **not suitable for binary search**.

Q10. What is Interpolation Search? How is it different from Binary Search?

✅ **Solution:**

- Works like **Binary Search** but instead of **halving**, it estimates the position using a formula:
  mid = low + ((key - arr[low]) * (high - low)) / (arr[high] - arr[low])

- **Best-case Complexity**: O(1)
- **Average-case Complexity**: O(log log n)
- **Worst-case Complexity**: O(n) (if elements are unevenly distributed).

Q13. What is Exponential Search? When is it useful?

✅ **Solution:**
Exponential Search is used when:

- The size of the array is **unknown or unbounded**.
- The array is **sorted**.

Steps:

1. Start at index **1**, **double** the index each time until you find a range where the key may exist.
2. Perform **Binary Search** in that range.

Time Complexity: O(log n)

Example Use Case: Searching in **infinite** or **large** datasets.

---

4. Miscellaneous Questions

Q14. What is Ternary Search? How does it compare to Binary Search?

✅ **Solution:**

- Ternary Search **divides** the array into **three** parts instead of two.
- **Best/Average-case Complexity:** $O(\log_3 n)$
- **Binary Search is faster** because $\log_2 n < \log_3 n$.

Q15. What is the time complexity of searching in a balanced BST?

✅ **Solution:**

- **Best/Average/Worst-case Complexity:** O(log n)
- Example: **AVL Tree**, **Red-Black Tree**.

Q16. What is the time complexity of searching in a Hash Table?

✅ **Solution:**

- **Best/Average-case:** O(1)
- **Worst-case:** O(n) (if many collisions occur).

Q17. Which searching algorithm is best for large, sorted datasets?

✅ **Solution:**

- **Binary Search (O(log n))** is generally the best.
- If **distribution is uniform**, **Interpolation Search (O(log log n))** is even better.

Q18. Which searching algorithm is best for small datasets?

✅ **Solution:**

- **Linear Search (O(n))** (no need for sorting overhead).

Sorting Algorithms

Q1. What is a sorting algorithm? Why is it important?

✅ **Solution:**
A **sorting algorithm** is a method used to **arrange elements** in a specific order (ascending or descending). Sorting is important because:

- It helps in **searching (Binary Search needs sorted data)**.
- It improves **data readability and organization**.
- It is widely used in **database indexing, scheduling, and optimization**.

Q2. What are the types of sorting algorithms?

✅ **Solution:**
Sorting algorithms are classified into:

1. **Comparison-based Sorting** (O(n log n) or $O(n^2)$)
    - Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort.
2. **Non-Comparison-based Sorting** (O(n))
    - Counting Sort, Radix Sort, Bucket Sort.

Q3. What is the difference between stable and unstable sorting algorithms?

✅ **Solution:**

- **Stable Sorting**: Maintains the relative order of duplicate elements.
- **Unstable Sorting**: Does **not** maintain the relative order of duplicate elements.

Algorithm	Stable?	Example
Bubble Sort	✅ Yes	[2,1,2',3] → [1,2,2',3]
Merge Sort	✅ Yes	[4,4',2,1] → [1,2,4,4']
Quick Sort	❌ No	[5,3,5',1] → [1,3,5',5]
Heap Sort	❌ No	[9,2,2',5] → [2',2,5,9]

Q4. What is the difference between in-place and out-of-place sorting?

✅ **Solution:**

- **In-place Sorting**: Uses **constant or O(1) extra space** (e.g., Quick Sort, Bubble Sort).
- **Out-of-place Sorting**: Requires **extra memory (O(n))** (e.g., Merge Sort, Counting Sort).

Q5. What are the best, worst, and average-case time complexities of sorting algorithms?

✅ **Solution:**

Algorithm	Best-case	Worst-case	Average-case	Space Complexity
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Quick Sort	O(n log n)	$O(n^2)$	O(n log n)	O(log n)
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)
Counting Sort	O(n + k)	O(n + k)	O(n + k)	O(k)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n + k)

Q6. Which sorting algorithms are best for small data sets?

✅ **Solution:**

- **Insertion Sort ($O(n^2)$)** (for nearly sorted small arrays).
- **Selection Sort ($O(n^2)$)** (if swapping cost is low).
- **Bubble Sort ($O(n^2)$)** (only when stability is needed).

Q7. Which sorting algorithms are best for large datasets?

✅ **Solution:**

- **Merge Sort (O(n log n))** (if stability is needed).
- **Quick Sort (O(n log n))** (if in-place sorting is needed).
- **Heap Sort (O(n log n))** (if guaranteed performance is needed).

---

2. Sorting Algorithm Comparisons

Q8. Why is Quick Sort faster than Merge Sort in practice?

✅ **Solution:**

- Quick Sort is **in-place (O(log n) space)**, while Merge Sort needs **O(n) extra space**.
- Quick Sort has **better cache performance** because it processes smaller subarrays in memory.

Q9. Why is Merge Sort preferred for linked lists over Quick Sort?

✅ **Solution:**

- Merge Sort does **not require random access**, making it better for **linked lists**.
- Quick Sort needs **extra pointer swaps**, which are costly for linked lists.

Q10. Why is Heap Sort not commonly used in practice?

✅ **Solution:**

- Heap Sort has **poor cache locality** compared to Quick Sort.
- It is **not stable** and requires **O(n) extra work** for heapify operations.

Q11. When is Counting Sort better than Quick Sort?

✅ **Solution:**

- When **input values are in a small range** (O(n + k) is faster than O(n log n)).
- Example: Sorting **grades of students (A, B, C, D)**.

Q12. What is the best sorting algorithm for nearly sorted data?

✅ **Solution:**

- **Insertion Sort (O(n))** performs better than O(n log n) algorithms.

- **Timsort (Python's sorting algorithm)** is optimized for nearly sorted data.

---

3. Advanced Sorting Concepts

Q13. What is the Time Complexity of Bubble Sort with an optimized approach?

✅ **Solution:**

- By **checking if swapping occurs**, we can stop early if the array is sorted.
- **Best-case complexity:** O(n).

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped:
            break
```

Q14. How does Radix Sort work? Why is it O(nk)?

✅ **Solution:**

- Radix Sort sorts **numbers digit-by-digit** from least significant to most significant.
- If numbers have **k digits**, it performs **k iterations of Counting Sort (O(n))**.
- Hence, total complexity = **O(nk)**.

Example: Sorting [170, 45, 75, 90]

1. Sort by **1's place** → [170, 90, 45, 75]
2. Sort by **10's place** → [170, 75, 90, 45]
3. Sort by **100's place** → [45, 75, 90, 170]

Q15. What is Timsort? Why is it used in Python?

✅ **Solution:**

- **Timsort = Merge Sort + Insertion Sort** (O(n log n)).

- It detects **already sorted parts** and optimizes performance.
- Python's sorted() function uses **Timsort** because it is **fast, stable, and efficient for real-world data**.

Q16. What is 3-way Quick Sort?

✅ **Solution:**

- **Normal Quick Sort**: Partitions into **two** parts.
- **3-way Quick Sort**: Partitions into **three** parts:
    1. Elements smaller than pivot.
    2. Elements equal to pivot.
    3. Elements greater than pivot.
- Useful for arrays with **many duplicate elements**.

Hashing

1. What is Hashing?

Solution: Hashing is a technique used to map data to a fixed-size table using a hash function. It provides efficient data retrieval in O(1) time complexity in the best case.

2. What is a Hash Function?

Solution: A hash function takes an input (key) and returns a fixed-size integer (hash value), which determines the index in a hash table.

3. What is a Hash Table?

Solution: A data structure that stores key-value pairs and uses a hash function to compute the index for storing values.

4. What is the need for Hashing?

Solution: Hashing is used for fast data retrieval in applications like databases, caching, and password storage.

5. What are the key properties of a good Hash Function?

Solution:

- **Deterministic** – The same input should always produce the same hash.
- **Uniform Distribution** – It should distribute keys evenly.
- **Efficient Computation** – It should be quick to compute.

- **Minimize Collisions** – It should reduce the chance of different inputs mapping to the same index.

Collision Handling

6. What is a Collision in Hashing?

Solution: A collision occurs when two different keys produce the same hash index.

7. What are the methods to resolve collisions in hashing?

Solution:

- **Chaining (Separate Chaining)**
- **Open Addressing (Linear Probing, Quadratic Probing, Double Hashing)**

8. What is Separate Chaining?

Solution: Uses linked lists at each index to store multiple elements that hash to the same location.

9. What is Open Addressing?

Solution: If a collision occurs, it finds another empty slot in the hash table.

10. What are the types of Open Addressing?

Solution:

- **Linear Probing:** Looks for the next available slot sequentially.
- **Quadratic Probing:** Uses a quadratic function to find the next slot.
- **Double Hashing:** Uses another hash function for better distribution.

Advanced Hashing Questions

11. What is Load Factor in Hashing?

Solution: Load Factor = (Number of elements in table) / (Size of the table).
It determines when to resize the hash table.

12. What is Rehashing?

Solution: When the load factor crosses a threshold, the table is resized, and all elements are rehashed into a new table.

13. How is Hashing used in Cryptography?

Solution: Hash functions like SHA-256 and MD5 are used to secure passwords and data integrity.

14. What is a Perfect Hash Function?

Solution: A function that produces unique hash values without collisions.

15. What is Universal Hashing?

Solution: A technique where a random hash function is chosen from a family of functions to reduce worst-case scenarios.

Applications of Hashing

16. How is Hashing used in Data Structures?

Solution: Hashing is used in hash tables, hash maps, and sets for fast data retrieval.

17. How is Hashing used in Databases?

Solution: Indexing techniques like Hash Indexing use hashing to speed up searches.

18. How is Hashing used in Caching?

Solution: Hash tables are used in caches like **LRU Cache** for quick lookup.

19. How does Hashing help in Password Security?

Solution: Hash functions like **bcrypt** store passwords securely with salting.

20. How is Hashing used in Load Balancing?

Solution: Consistent hashing distributes requests across servers efficiently.

Comparison & Performance

21. How does Hashing differ from Binary Search Trees (BSTs)?

Solution:

- Hashing has O(1) average time complexity, whereas BSTs have O(log n).
- Hashing is unordered, whereas BSTs maintain order.

22. What is the Worst-Case Time Complexity of Hashing?

Solution: O(n) if all elements hash to the same index (poor hash function).

23. How does Double Hashing work?

Solution: A secondary hash function determines the step size to resolve collisions.

24. What is Rolling Hashing?

Solution: Used in string matching algorithms (e.g., **Rabin-Karp Algorithm**) to compute hash values efficiently for substrings.

25. What is the difference between HashMap and HashSet?

Solution:

- **HashMap** stores key-value pairs.
- **HashSet** only stores unique keys.

Prefix Sum

Q1. What is Prefix Sum? Why is it useful?

✅ **Solution:**

- **Prefix Sum** is a technique where we precompute the **cumulative sum** of an array to quickly answer range sum queries.
- It helps in **reducing time complexity** from **O(n) to O(1)** for sum queries.

Example:
For an array **arr = [3, 1, 4, 8, 6]**, the **Prefix Sum array (PSA)** is:
3,4,8,16,223, 4, 8, 16, 223,4,8,16,22
This allows us to compute sum in **O(1)**:
Sum of range **[1, 3]** → prefix[3] - prefix[0] = 16 - 3 = 13.

Q2. How do you construct a Prefix Sum array? What is its time complexity?

✅ **Solution:**
The **Prefix Sum array** is built using the formula:

prefix[i]=prefix[i−1]+arr[i]prefix[i] = prefix[i - 1] + arr[i]prefix[i]=prefix[i−1]+arr[i]

Time Complexity: O(n) (single pass).
Space Complexity: O(n) (extra array).

Python Code:

```
def prefix_sum(arr):
    n = len(arr)
    prefix = [0] * n
    prefix[0] = arr[0]
    for i in range(1, n):
        prefix[i] = prefix[i - 1] + arr[i]
    return prefix
```

Q3. How does Prefix Sum improve range sum queries compared to a naive approach?

✅ **Solution:**

Approach	Time Complexity	Explanation
Naive (Looping)	O(n)	Iterates over the range for sum.
Prefix Sum	O(1)	Uses precomputed array: **prefix[j] - prefix[i-1]**.

💡 **Example:**
Array: [2, 4, 6, 8, 10]
Prefix Sum: [2, 6, 12, 20, 30]
Range sum **[1,3]** → prefix[3] - prefix[0] = 20 - 2 = 18.

Q4. What are some real-world applications of Prefix Sum?

✅ **Solution:**

1. **Range Sum Queries** (Competitive Programming).
2. **Difference Arrays** (Efficient updates on ranges).
3. **Image Processing** (Integral images for fast calculations).
4. **Game Development** (Fast collision detection).
5. **Text Analysis** (Finding frequency of words efficiently).

Q5. What are the limitations of Prefix Sum? How can they be solved?

✅ **Solution:**

Limitation	Solution
Cannot handle **dynamic updates** efficiently	Use **Fenwick Tree / Segment Tree**.
Extra **O(n) space** for prefix array	Use **in-place modification if needed**.
Works only for **sum queries**	Extend to **XOR prefix sum, GCD prefix sum, etc.**.

Heap

1. What is a Heap?

Solution: A **Heap** is a specialized tree-based data structure that satisfies the **heap property**:

- In a **Max Heap**, the parent node is always greater than or equal to its children.
- In a **Min Heap**, the parent node is always smaller than or equal to its children.

2. What are the types of Heaps?

Solution:

- **Min Heap**: The root node has the smallest value.
- **Max Heap**: The root node has the largest value.
- **Binary Heap**: A complete binary tree that follows the heap property.
- **Fibonacci Heap**: A collection of min-heaps used in advanced applications.
- **Binomial Heap**: A set of binomial trees with specific merging properties.

3. What is a Binary Heap?

Solution: A **Binary Heap** is a **complete binary tree** that maintains the **heap property**.

4. Why is a Heap called a Complete Binary Tree?

Solution: A Heap is always **complete**, meaning:

- Every level is filled except possibly the last.
- The last level is filled from left to right.

5. What are the applications of Heaps?

Solution:

- **Priority Queues** (Scheduling algorithms, Dijkstra's Algorithm)
- **Heap Sort**
- **Finding K largest or smallest elements**
- **Graph Algorithms** (Prim's, Dijkstra's Algorithm)
- **Memory Management (Garbage Collection)**
- **Load Balancing in Servers**

Heap Operations

6. What are the basic operations of a Heap?

Solution:

- **Insert (O(log n))**: Adds a new element while maintaining heap property.
- **Delete (O(log n))**: Removes an element, typically the root.
- **Heapify (O(log n))**: Rearranges elements to maintain the heap property.
- **Extract Min/Max (O(log n))**: Removes and returns the root of the heap.
- **Build Heap (O(n))**: Constructs a heap from an unordered array.

7. How is a Heap stored in an Array?

Solution: A Heap is usually implemented as an **array** with:

- **Parent at index** i

- **Left child at index** 2i + 1
- **Right child at index** 2i + 2

8. What is Heapify?

Solution: Heapify is the process of maintaining the **heap property** after inserting or deleting an element.

9. What is Build Heap?

Solution: The **Build Heap** algorithm constructs a heap from an unsorted array in **O(n) time**.

10. What is the time complexity of Heap operations?

Solution:

- **Insert → O(log n)**
- **Delete → O(log n)**
- **Heapify → O(log n)**
- **Build Heap → O(n)**
- **Heap Sort → O(n log n)**

Heap Implementation & Variations

11. How is Insert operation performed in a Heap?

Solution:

1. Insert the new element at the end of the heap.
2. **Bubble up** (swap with the parent if necessary) to maintain the heap property.

12. How is Delete operation performed in a Heap?

Solution:

1. Swap the root with the last element.
2. Remove the last element.
3. **Heapify down** to maintain the heap property.

13. How does Heap Sort work?

Solution:

1. Build a heap from the input array (**O(n)**).
2. Extract the max/min element and place it at the end of the array (**O(n log n)**).
3. Repeat until the heap is empty.

14. What is the difference between Min Heap and Max Heap?

Solution:

- **Min Heap**: The smallest element is at the root.
- **Max Heap**: The largest element is at the root.

15. What is the difference between Heap and Stack?

Solution:

- **Heap (Data Structure)**: A tree-based structure for sorting and priority queues.
- **Heap (Memory Management)**: Dynamically allocated memory storage.
- **Stack**: Follows LIFO (Last In, First Out) for function calls and local variables.

LinkedList Basics

1. **What is a linked list?**
   Solution: A linked list is a linear data structure in which elements (nodes) are stored in non-contiguous memory locations. Each node consists of data and a pointer/reference to the next node.
2. **What are the types of linked lists?**
   Solution:

   - Singly Linked List
   - Doubly Linked List
   - Circular Linked List
   - Circular Doubly Linked List
3. **What are the key differences between arrays and linked lists?**
   Solution:
   - Arrays have fixed size, while linked lists are dynamic.
   - Arrays support random access (O(1)), while linked lists require sequential traversal (O(n)).
   - Insertion/deletion is easier in linked lists (O(1) at the head).
4. **What are the advantages of linked lists?**
   Solution:
   - Dynamic memory allocation
   - Efficient insertions and deletions
   - No memory wastage due to predefined sizes

5. **What are the disadvantages of linked lists?**
   Solution:
   - Extra memory for pointers
   - Slower access time due to sequential traversal

Linked List Operations

6. **What operations can be performed on a linked list?**
   Solution:
   - Insertion (beginning, middle, end)
   - Deletion (beginning, middle, end)
   - Searching
   - Reversal
7. **What is the time complexity for inserting a node in a linked list?**
   Solution:
   - **At the beginning:** O(1)
   - **At the end:** O(n) (without tail pointer), O(1) (with tail pointer)
   - **At a specific position:** O(n)
8. **How is deletion performed in a linked list?**
   Solution:
   - If deleting the head node, update the head pointer.
   - If deleting a middle node, traverse to the previous node and update the next pointer.
   - If deleting the last node, update the second-last node's next pointer to NULL.
9. **How do you find the length of a linked list?**
   Solution:
   - Start with a counter at 0.
   - Traverse through each node, increasing the counter until reaching NULL.
10. **What is the time complexity of searching for an element in a linked list?**
    Solution:
    - **Best case:** O(1) (if the element is at the head).
    - **Worst case:** O(n) (if the element is at the end).

Doubly and Circular Linked Lists

11. What is a doubly linked list?
Solution: A doubly linked list has two pointers:

- next → Points to the next node.
- prev → Points to the previous node.

12. What are the advantages of a doubly linked list?
Solution:

- Can traverse in both directions.
- Efficient deletion from both ends.

13. What is a circular linked list?
Solution: A circular linked list has its last node pointing back to the first node, forming a cycle.

14.What are the applications of circular linked lists?
Solution:

- Operating system scheduling (Round Robin).
- Multiplayer gaming applications.

15.How do you check if a linked list is circular?
Solution: If the last node's next pointer is not NULL but points back to the head, it's circular.

Memory Management and Edge Cases

16. How is memory allocated in a linked list?
Solution: Memory is allocated dynamically using malloc() in C or new in C++.

17. What happens if we delete a node without freeing memory?
Solution: It leads to a memory leak, as memory remains occupied but is no longer accessible.

18. What happens if we try to access a deleted node?
Solution: It causes undefined behaviour or segmentation faults.

19. What is the difference between singly and doubly linked lists in terms of deletion?
Solution: In a singly linked list, we must traverse to the previous node, while in a doubly linked list, we can directly access the previous node.

20. Can a linked list be implemented without pointers?
Solution: Yes, using arrays and indices as pointers.

Stack & Queue

Q1. What is a Stack? Explain its working principle.

✅ **Solution:**
A **Stack** is a **linear data structure** that follows the **LIFO (Last In, First Out)** principle.

- **Push(x):** Inserts element at the **top**.
- **Pop():** Removes the **top** element.

- **Peek():** Returns **top** element without removing.
- **isEmpty():** Checks if the stack is empty.

💡 **Example:**

Push(10) → [10]

Push(20) → [10, 20]

Pop() → 20 removed → [10]

Q2. What are the real-life applications of Stacks?

✅ **Solution:**

1. **Function Calls (Recursion)** – Uses **stack memory**.
2. **Undo/Redo operations** – Text editors.
3. **Browser History** – Back & Forward navigation.
4. **Balancing Parentheses** – Validates expressions.
5. **Expression Evaluation** – Infix to Postfix conversion.

Q3. What is Stack Overflow and Stack Underflow?

✅ **Solution:**

- **Stack Overflow:** When trying to push into a **full** stack.
- **Stack Underflow:** When trying to pop from an **empty** stack.

Example:

Stack size = 3

Push(1), Push(2), Push(3) → FULL

Push(4) → ❌ Stack Overflow

Pop() 3 times → EMPTY

Pop() again → ❌ Stack Underflow

Q4. What is the time complexity of Stack operations?

✅ **Solution:**

Operation	Complexity
Push	O(1)
Pop	O(1)
Peek	O(1)
isEmpty	O(1)

Q5. What is the use of Stacks in Expression Evaluation?

✅ **Solution:**
Used for **Infix to Postfix conversion** and **Postfix evaluation** using **two stacks**.

Example:
Infix: (2 + 3) * 5
Postfix: 2 3 + 5 *

Q6. What is a Min Stack and how does it work?

✅ **Solution:**
A **Min Stack** allows retrieving the **minimum element in O(1)**.

📌 **QUEUE - Theory Questions & Solutions**

Q7. What is a Queue? Explain its working principle.

✅ **Solution:**
A **Queue** follows **FIFO (First In, First Out)**.

- **Enqueue(x):** Insert at **rear**.
- **Dequeue():** Remove from **front**.

💡 **Example:**

Enqueue(10) → [10]

Enqueue(20) → [10, 20]

Dequeue() → 10 removed → [20]

Q10. What are the types of Queues?

✅ **Solution:**

1. **Simple Queue** – FIFO behavior.
2. **Circular Queue** – Connects rear → front.

3. **Double-Ended Queue (Deque)** – Insert/Delete at **both ends**.
4. **Priority Queue** – Dequeues elements based on **priority**.

Q11. What are real-life applications of Queues?

✅ **Solution:**

1. **CPU Scheduling** – Process execution order.
2. **Printers** – Print job scheduling.
3. **Call Center Systems** – Customer service queues.
4. **Graph Traversal (BFS)** – Uses **Queue**.

Q12. What is the difference between Stack and Queue?

✅ **Solution:**

Feature	Stack (LIFO)	Queue (FIFO)
Insertion	Push (Top)	Enqueue (Rear)
Deletion	Pop (Top)	Dequeue (Front)
Access Order	Last In, First Out	First In, First Out
Example	Function Call Stack	Print Jobs Queue

Q13. What is a Circular Queue and why is it useful?

✅ **Solution:**
A **Circular Queue** connects **rear → front** to reuse **memory efficiently**.

- In a **linear queue**, dequeue() leads to **wasted space**.

Q14. What is a Deque (Double-Ended Queue)?

✅ **Solution:**
A **Deque** allows **insertion and deletion** from **both ends**.

Bitwise Operations

📌 1. What are Bitwise Operators in C/C++/Java/Python?

✅ **Solution:**
Bitwise operators **directly manipulate bits** in binary representation. The common operators include:

Operator	Symbol	Description
AND	&	1 & 1 = 1, else 0
OR	`	`
XOR	^	1 ^ 1 = 0, 0 ^ 1 = 1
NOT	~	Inverts bits (~5 = -6 in 2's complement)
Left Shift	<<	a << n = a * 2^n
Right Shift	>>	a >> n = a / 2^n

💡 **Example:**

```
a = 5  # 0101
b = 3  # 0011

print(a & b)  # 1  (0001)
print(a | b)  # 7  (0111)
print(a ^ b)  # 6  (0110)
print(~a)     # -6 (2's complement: 1010)
print(a << 1) # 10 (1010)
print(a >> 1) # 2  (0010)
```

📌 2. What are the applications of Bitwise Operators in DSA?

✅ **Solution:**
Bitwise operations help in:

- **Optimizing calculations** (Multiplication, Division by 2).
- **Checking if a number is even/odd** (n & 1 == 0 for even).
- **Finding unique elements in an array** (using XOR).
- **Efficiently computing powers of 2** (1 << n = 2^n).
- **Fast swapping of variables** (a = a ^ b; b = a ^ b; a = a ^ b).
- **Checking if a number is a power of 2** (n & (n-1) == 0).

📌 3. How to check if a number is even or odd using Bitwise Operators?

✅ **Solution:**

```
def is_even(n):
    return (n & 1) == 0


print(is_even(10))  # True
print(is_even(7))   # False
```

👉 **Explanation:**

- n & 1 checks if the **last bit** is 0 (even) or 1 (odd).

📌 4. How to swap two numbers without using a temporary variable?

✅ **Solution:**
Using XOR:

```
def swap(a, b):
    a = a ^ b
    b = a ^ b
    a = a ^ b
    return a, b


x, y = swap(5, 7)
print(x, y)  # Output: 7 5
```


👉 **Explanation:**

- a ^ b stores the **combined** bit information.
- XOR again restores the original values.

📌 5. How to find the only non-repeating element in an array where every other element appears twice?

✅ **Solution:**
Use XOR:

```
def find_unique(arr):
```

```
    res = 0
    for num in arr:
        res ^= num
    return res

print(find_unique([2, 3, 2, 4, 3, 5, 4]))  # Output: 5
```

👉 **Explanation:**

- x ^ x = 0 cancels out duplicate numbers, leaving only the **unique number**.

📌 **6. How to check if a number is a power of 2 using Bitwise Operations?**

✅ **Solution:**

```
def is_power_of_2(n):
    return n > 0 and (n & (n - 1)) == 0

print(is_power_of_2(8))  # True
print(is_power_of_2(10)) # False
```


👉 **Explanation:**

- **Power of 2 numbers** have only **one bit set** (1000, 10000).
- n & (n - 1) == 0 removes the rightmost 1, so it should become 0.

📌 **7. How to count the number of set bits (1s) in an integer?**

✅ **Solution:**
Using **Brian Kernighan's Algorithm**:

```
def count_set_bits(n):
    count = 0
    while n:
        n &= (n - 1)  # Clears the rightmost set bit
        count += 1
    return count

print(count_set_bits(15))  # Output: 4 (1111)
```

👉 **Explanation:**

- Each n & (n - 1) **removes the last set bit**, counting bits in **O(log n)** time.

📌 **8. How to reverse bits of a number?**

✅ **Solution:**

```
def reverse_bits(n):
    result = 0
    for _ in range(32):  # For 32-bit integer
        result = (result << 1) | (n & 1)
        n >>= 1
    return result

print(bin(reverse_bits(5)))  # Output: 0b10100000000000000000000000000000
```

👉 **Explanation:**

- We **shift left** the result and **add the last bit** of n using |.

📌 **9. How to find the rightmost set bit in an integer?**

✅ **Solution:**

```
def rightmost_set_bit(n):
    return n & -n  # Gives position of rightmost set bit

print(bin(rightmost_set_bit(10)))  # Output: 0b10 (2 in decimal)
```

👉 **Explanation:**

- n & -n isolates the **rightmost 1 bit** using **2's complement**.

📌 **10. How to find XOR of numbers from 1 to n efficiently?**

✅ **Solution:**

```
def xor_n(n):
    if n % 4 == 0:
        return n
```

```
    elif n % 4 == 1:
        return 1
    elif n % 4 == 2:
        return n + 1
    else:
        return 0

print(xor_n(10))  # Output: 10
```

👉 **Explanation:**
The XOR pattern for 1 to n repeats every 4 numbers:

n XOR(1 to n)

1 1

2 3

3 0

4 4

📌 **11. How to multiply a number by 7 using Bitwise Operators?**

✅ **Solution:**

```
def multiply_by_7(n):
    return (n << 3) - n  # (n * 8) - n = n * 7

print(multiply_by_7(5))  # Output: 35
```

👉 **Explanation:**

- n << 3 = n * 8, then subtract n to get n * 7.

Tree

1. What is a tree in data structures?

Answer:

A **tree** is a **non-linear data structure** consisting of **nodes** where each node has a **parent-child** relationship. It starts with a **root node**, and each node can have multiple child nodes, forming a hierarchical structure.

---

2. What are the different types of trees in DSA?

Answer:

- **Binary Tree**: Each node has at most 2 children.
- **Binary Search Tree (BST)**: A binary tree where the left child is smaller, and the right child is greater than the parent node.
- **Balanced Binary Tree**: A tree where the height difference between left and right subtrees is at most 1.
- **Complete Binary Tree**: All levels are completely filled except possibly the last, which is filled from left to right.
- **Full Binary Tree**: Every node has either 0 or 2 children.
- **Perfect Binary Tree**: All internal nodes have exactly 2 children, and all leaf nodes are at the same level.
- **AVL Tree**: A self-balancing BST where the difference between heights of left and right subtrees is at most 1.
- **Red-Black Tree**: A self-balancing BST that ensures logarithmic height.

---

3. What is the difference between a Binary Tree and a Binary Search Tree (BST)?

Answer:

Feature	Binary Tree	Binary Search Tree (BST)
Structure	Any arrangement of nodes	Ordered: Left < Parent < Right
Searching	O(n) in worst case	O(log n) in balanced BST
Insertion	No order required	Must maintain order

---

4. What is tree traversal? Name its types.

Answer:

Tree traversal is a method to visit all nodes of a tree in a specific order.
Types:

1. **Depth First Search (DFS)**
    - **Preorder (Root → Left → Right)**

- **Inorder (Left → Root → Right)**
- **Postorder (Left → Right → Root)**

2. **Breadth First Search (BFS) / Level Order Traversal**

---

5. What is the height of a tree?

Answer:

The **height of a tree** is the length of the longest path from the **root node** to a **leaf node**.

Formula for a binary tree with **n** nodes:

Height=log⁡2(n) (for balanced trees)\text{Height} = \log_2(n) \text{ (for balanced trees)}Height=log2(n) (for balanced trees)

---

6. What is the time complexity of tree traversals?

Answer:

- **DFS (Preorder, Inorder, Postorder):** O(n)
- **BFS (Level Order):** O(n)
- **Searching in BST:** O(log n) (for balanced BST), O(n) (for skewed BST)

---

7. How to find the lowest common ancestor (LCA) of two nodes in a Binary Tree?

Answer:

- If **one node is in the left subtree** and the **other is in the right**, the current node is the LCA.
- If **both nodes are in the left** subtree, LCA is in the left subtree.
- If **both are in the right**, LCA is in the right subtree.

---

8. What is the difference between BFS and DFS?

Answer:

Feature	**BFS (Level Order)**	**DFS (Preorder, Inorder, Postorder)**
Traversal	Level by level	Depth-first, one branch at a time
Data Structure Used	Queue	Stack (or recursion)
Time Complexity	O(n)	O(n)
Space Complexity	O(n)	O(h), where h is the height

---

9. What is an AVL Tree?

Answer:

An **AVL Tree** is a self-balancing **Binary Search Tree (BST)** where the height difference of left and right subtrees (Balance Factor) is at most **1** for all nodes.

---

10. What are rotations in AVL Trees?

Answer:

Rotations help in balancing an **AVL Tree**.
Types:

- **LL Rotation (Right Rotation)**
- **RR Rotation (Left Rotation)**
- **LR Rotation (Left-Right Rotation)**
- **RL Rotation (Right-Left Rotation)**

---

11. What is a Red-Black Tree?

Answer:

A **Red-Black Tree** is a self-balancing BST where each node follows these rules:

- Each node is **either red or black**.
- Root is always **black**.
- Red nodes **cannot have consecutive red children**.
- Every path from root to null must have the same number of **black nodes**.

Time Complexity: **O(log n) for insertion, deletion, search**.

---

12. What is a Heap Tree?

Answer:

A **Heap** is a **complete binary tree** that follows the **heap property**:

- **Max Heap:** Parent node is **greater** than its children.
- **Min Heap:** Parent node is **smaller** than its children.

Used in **Priority Queues** and **Heap Sort**.

---

13. What is Morris Traversal?

Answer:

Morris Traversal is a way to **traverse a Binary Tree without recursion or stack** by modifying the tree temporarily.

---

14. What is the difference between a BST and a Heap?

Answer:

Feature	Binary Search Tree (BST)	Heap
Order	Left < Root < Right	Parent > Children (Max Heap)
Searching	O(log n)	O(n)
Insertion	O(log n)	O(log n)
Usage	Fast searching	Priority Queues, Heap Sort

---

15. What is a Trie?

Answer:

A **Trie (Prefix Tree)** is a tree used for **storing and searching strings efficiently**. Each node represents a character, and words are stored as paths.

Used for:

- **Auto-complete**
- **Spell check**
- **IP routing**

---

16. What is a Segment Tree?

Answer:

A **Segment Tree** is a **binary tree** used for **range queries** (sum, min, max) in O(log n) time.
Example: Finding the sum of elements in an array range **[L, R]**.

---

17. What is a B-Tree?

Answer:

A **B-Tree** is a **self-balancing search tree** used in databases and file systems, where each node can have multiple children.

Properties:

- **All leaves are at the same level.**
- **Each node can store multiple keys.**

---

18. What is a Ternary Tree?

Answer:

A **Ternary Tree** is a tree where each node can have up to **three** children.

---

19. What is a Threaded Binary Tree?

Answer:

A **Threaded Binary Tree** is a BST where **null pointers are replaced with pointers to in-order successor or predecessor**.

---

20. What is a Splay Tree?

Answer:

A **Splay Tree** is a **self-adjusting BST** that moves the most recently accessed node to the root, improving performance for frequently accessed elements.