

Most Frequently asked DSA questions in MAANG

DSA Questions

1. Two Sum

Problem: Given an array of integers `nums` and an integer `target`, return indices of the two numbers such that they add up to `target`. Assume each input has exactly one solution, and you may not use the same element twice.

Example:

```
def twoSum(nums, target):  
    hashmap = {}  
    for i, num in enumerate(nums):  
        if target - num in hashmap:  
            return [hashmap[target - num], i]  
        hashmap[num] = i
```

Example

```
print(twoSum([2,7,11,15], 9)) # [0,1]
```

Explanation:

- Use a hashmap to store elements and their indices.
- For each number, check if `target - num` already exists in the hashmap.
- If yes, return their indices.
- **Time:** $O(n)$
- **Space:** $O(n)$

2. Best Time to Buy and Sell Stock

Problem: You are given an array `prices` where `prices[i]` is the price of a stock on day `i`. Find the maximum profit you can achieve by choosing a day to buy and a later day to sell.

```
def maxProfit(prices):  
    min_price = float('inf')  
    max_profit = 0  
    for price in prices:  
        min_price = min(min_price, price)  
        max_profit = max(max_profit, price - min_price)  
    return max_profit
```

Example

```
print(maxProfit([7,1,5,3,6,4])) # 5
```

- Track the minimum stock price so far.
- At each step, calculate profit = price - min_price.
- Update maximum profit accordingly.
- **Time:** $O(n)$
- **Space:** $O(1)$

3. Valid Parentheses

Problem: Given a string containing only `()[]{}` , determine if the string is valid (every open bracket has a matching closing bracket in correct order).

Example:

```
def isValid(s):
    stack = []
    mapping = {'(': ')', '[': ']', '': '{'}
    for char in s:
        if char in mapping:
            top = stack.pop() if stack else '#'
            if mapping[char] != top:
                return False
        else:
            stack.append(char)
    return not stack
```

Example

```
print(isValid("()[]{}")) # True
```

Explanation:

- Use a stack to track opening brackets.
- Every time a closing bracket appears, pop and check if it matches.
- If mismatched or stack isn't empty at end → invalid.
- **Time:** $O(n)$
- **Space:** $O(n)$

4. Maximum Subarray (Kadane's Algorithm)

Problem: Find the contiguous subarray with the largest sum and return the sum.

```
def maxSubArray(nums):
    max_sum = nums[0]
    curr_sum = nums[0]
    for num in nums[1:]:
        curr_sum = max(num, curr_sum + num)
        max_sum = max(max_sum, curr_sum)
    return max_sum

# Example
print(maxSubArray([-2,1,-3,4,-1,2,1,-5,4])) # 6

print(longest_common_prefix(["flower", "flow", "flight"]))
```

Explanation:

- At each index, either extend the previous subarray or start new.
- Keep track of global maximum.
- Time: $O(n)$
- Space: $O(1)$

5. Merge Intervals

Problem: Given a collection of intervals, merge all overlapping intervals.

```
def merge(intervals):
    intervals.sort(key=lambda x: x[0])

    merged = [intervals[0]]

    for start, end in intervals[1:]:
        if start <= merged[-1][1]:
            merged[-1][1] = max(merged[-1][1], end)
        else:
            merged.append([start, end])

    return merged
```

- Sort intervals by start.
- Merge overlapping intervals by updating end time.
- Add new interval only when no overlap.
- **Time:** $O(n \log n)$ (sorting)
- **Space:** $O(n)$ (result list)

6. Product of Array Except Self

Problem: Return array where each element is product of all other elements (no division).

Example: $[1,2,3,4] \rightarrow [24,12,8,6]$

Approach: Prefix and suffix products in two passes.

```
def product_except_self(nums):
```

```
    n = len(nums)
```

```
    res = [1]*n
```

```
    prefix = 1
```

```
    for i in range(n):
```

```
        res[i] = prefix
```

```
        prefix *= nums[i]
```

```
    suffix = 1
```

```
    for i in range(n-1, -1, -1):
```

```
        res[i] *= suffix
```

```
        suffix *= nums[i]
```

```
    return res
```

```
print(product_except_self([1,2,3,4])) # [24,12,8,6]
```

Complexity: $O(n)$, $O(1)$ extra (output not counted)

7. Group Anagrams

Problem: Given an array of strings, group the anagrams together..

```
from collections import defaultdict
```

```
def groupAnagrams(strs):
    groups = defaultdict(list)
    for word in strs:
        key = "".join(sorted(word))
        groups[key].append(word)
    return list(groups.values())
```

Example

```
print(groupAnagrams(["eat", "tea", "tan", "ate", "nat", "bat"]))
# [['eat', 'tea', 'ate'], ['tan', 'nat'], ['bat']]
```

- Sort each word → use as a key (anagrams share the same sorted form).
- Group words by key.
- **Time:** $O(n \cdot k \log k)$ (n = words, k = length of each word)
- **Space:** $O(n \cdot k)$

8. Longest Substring Without Repeating Characters

Problem: Given a string s , find the length of the longest substring without repeating characters.

Example: $\text{nums}=[1,1,1]$, $k=2 \rightarrow 2$

Approach: Prefix sums + hashmap counting occurrences.

```
def lengthOfLongestSubstring(s):
    seen = {}
    left = 0
    max_len = 0
    for right, char in enumerate(s):
        if char in seen and seen[char] >= left:
            left = seen[char] + 1
        seen[char] = right
        max_len = max(max_len, right - left + 1)
    return max_len
```

Example

```
print(lengthOfLongestSubstring("abcabcbb")) # 3 ("abc")
```

9. Word Search (Backtracking)

Problem: Given an $m \times n$ board and a word, return true if the word exists in the grid.

The word can be constructed from adjacent cells (horizontally/vertically), and each cell can be used once.

```
def exist(board, word):
    rows, cols = len(board), len(board[0])

    def dfs(r, c, i):
        if i == len(word): return True
        if r < 0 or c < 0 or r >= rows or c >= cols or board[r][c] != word[i]:
            return False

        temp, board[r][c] = board[r][c], "#"
        found = (dfs(r+1,c,i+1) or dfs(r-1,c,i+1) or
                 dfs(r,c+1,i+1) or dfs(r,c-1,i+1))
        board[r][c] = temp
        return found

    for r in range(rows):
        for c in range(cols):
            if dfs(r,c,0): return True
    return False

# Example
board = [
    ["A","B","C","E"],
    ["S","F","C","S"],
    ["A","D","E","E"]
]
print(exist(board, "ABCCED")) # True
```

Explanation:

- Use DFS + backtracking to explore neighbors.
- Mark visited cell temporarily (#) to avoid reuse.
- Return true if full word matched.
- Time: $O(m \cdot n \cdot 4^L)$ (L = word length)
- Space: $O(L)$ recursion stack

10. Linked List Cycle Detection (Floyd's Algorithm)

Problem: Given the head of a linked list, determine if the linked list has a cycle in it.

```
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

def hasCycle(head):
    slow, fast = head, head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    if slow == fast:
        return True
    return False
```

Explanation:

- Use **two pointers**: slow moves 1 step, fast moves 2 steps.
- If cycle exists, they will eventually meet.
- If fast reaches None, no cycle.
- **Time:** $O(n)$
- **Space:** $O(1)$

11. Minimum Window Substring

Problem: Given strings *s* and *t*, find minimum window in *s* containing all chars of *t*. Return "" if none.

Example: *s*="ADOBECODEBANC", *t*="ABC" → "BANC"

Approach: Sliding window + counts of required chars; expand then contract. A CTE (Common Table Expression) is a temporary, named result set that you can reference within a SQL query. It improves readability and simplifies complex subqueries or recursive logic.

Syntax:

```
from collections import Counter
```

```
def min_window(s, t):
    need = Counter(t)
    missing = len(t)
    left = 0
    best = (0, float('inf'))
    for right, ch in enumerate(s):
        if need[ch] > 0:
            missing -= 1
        need[ch] -= 1
        while missing == 0:
            if right - left < best[1] - best[0]:
                best = (left, right)
            # try to move left
            need[s[left]] += 1
            if need[s[left]] > 0:
                missing += 1
            left += 1
    l, r = best
    return "" if r == float('inf') else s[l:r+1]
```

```
print(min_window("ADOBECODEBANC", "ABC")) # "BANC"
```

Benefits:

- **Complexity:** $O(|s| + |t|)$ typical, $O(1)$ alphabet assumption.
Tip: Explaining need and missing succinctly is crucial in interviews.

12. Longest Subarray with At Most K Distinct

Problem: Given array (or string) find longest subarray with at most k distinct elements.

Example: [1,2,1,2,3], k=2 → length 4 (1,2,1,2)

Approach: Sliding window + hashmap counting distinct in window.

```
from collections import defaultdict

def longest_at_most_k_distinct(nums, k):
    cnt = defaultdict(int)
    left = 0
    distinct = 0
    best = 0
    for right, x in enumerate(nums):
        if cnt[x] == 0:
            distinct += 1
        cnt[x] += 1
        while distinct > k:
            cnt[nums[left]] -= 1
            if cnt[nums[left]] == 0:
                distinct -= 1
            left += 1
        best = max(best, right - left + 1)
    return best

print(longest_at_most_k_distinct([1,2,1,2,3], 2)) # 4
```

Explanation:

- **Complexity:** $O(n)$.

Graph Traversal

13. Top K Frequent Elements

Problem: Return the k most frequent elements from an integer array.

```
from collections import deque
def import heapq
from collections import Counter
```

```
def topKFrequent(nums, k):
    count = Counter(nums)
    return [x for x, _ in
            heapq.nlargest(k,
                           count.items(), key=lambda
                               x: x[1])]

```

Example

```
print(topKFrequent([1,1,1,2,2,3]
, 2)) # [1,2]
```

Example adjacency

```
adj = {1:[2,3], 2:[4], 3:[], 4:[]}
print(bfs_shortest_path(adj, 1, 4)) # [1,2,4]
```

Count frequency using hashmap.

Use max-heap to extract top k frequent elements.

Time: $O(n \log k)$

Space: $O(n)$

14. Detect Cycle in Directed Graph (DFS)

Problem: Detect if a directed graph has a cycle.

Approach: DFS with 3-color marking (0=unvisited,1=visiting,2=visited).

```
def has_cycle_directed(adj):
    nstates = {}
    def dfs(u):
        if nstates.get(u,0) == 1:
            return True
        if nstates.get(u,0) == 2:
            return False
        nstates[u] = 1
        for v in adj.get(u, []):
            if dfs(v):
                return True
        nstates[u] = 2
        return False
    for node in adj:
        if nstates.get(node,0) == 0:
            if dfs(node):
                return True
    return False

print(has_cycle_directed({1:[2], 2:[3], 3:[1]}))
```

Explanation:

- **Complexity:** $O(V+E)$..
- WHERE t.customer_id IS NULL ensures the customer had **no purchase in the last 6 months**.

15. Number of Connected Components (Union-Find)

Problem: Given n nodes and edge list, count connected components.

Approach: Union-Find (disjoint set union).Using IS NULL / IS NOT NULL:

```
SELECT * FROM employees WHERE manager_id IS NULL;
```

```
class DSU:
    def __init__(self, n):
        self.par = list(range(n))
        self.rank = [0]*n
    def find(self, a):
        while self.par[a] != a:
            self.par[a] = self.par[self.par[a]]
            a = self.par[a]
```

```

    return a
def union(self, a, b):
    ra, rb = self.find(a), self.find(b)
    if ra == rb:
        return False
    if self.rank[ra] < self.rank[rb]:
        ra, rb = rb, ra
    self.par[rb] = ra
    if self.rank[ra] == self.rank[rb]:
        self.rank[ra] += 1
    return True

def count_components(n, edges):
    dsu = DSU(n)
    for a, b in edges:
        dsu.union(a, b)
    roots = set(dsu.find(i) for i in range(n))
    return len(roots)

print(count_components(5, [(0,1),(1,2),(3,4)])) # 2

```

Complexity: $\sim O(\alpha(n))$ per op.

Tip: Explain path compression + union by rank.

PDF Owner Bhavesh Arora : [Bhavesh Arora | LinkedIn](#)

