IBM Coding Questions with Answers 2024.

1. **How to find the maximum subarray sum in an array of integers.**

```
def max_subarray_sum(a):
  max_so_far = 0
max_ending_here = 0
for i in range(len(a)):
  max_ending_here = max(0, max_ending_here + a[i])
max_so_far = max(max_so_far, max_ending_here)
return max_so_far
```

To find the maximum subarray sum in an array of integers we have tackled two main variables which are : **max_so_far** and **max_ending_here**. Both the elements represent the maximum subarray that has been going and the sum that ends at the current index. The algorithm here stats with initializing **max_so_far** to 0 then iterates keeping track of **max_ending_here**. After the iterating process is completed it sends the value of

max_so_far.

2. **A simple method to sort an array of integers in ascending and descending order**

Python

```
def bubble_sort(a):
    for i in range(len(a) - 1):
        for j in range(len(a) - i - 1):
            if a[j] > a[j + 1]:
                a[j], a[j + 1] = a[j + 1], a[j]
```

The algorithm here works by consistently comparing the adjacent components which are present in the array and exchanging them to get them in the right order. The algorithm starts and iterates over it, comparing each component to its next one. If the present component is greater than its side one the algorithm exchanges them. The algorithm keeps repeating again and again until it's sorted.

3. **The easiest way to find the shortest path between two nodes in a graph**

Python

```
def dijkstra(graph, source):

    distances = {}

    for node in graph:

        distances[node] = float('inf')

    distances[source] = 0

    queue = [source]

    while queue:

        current_node = queue.pop(0)

        for neighbor in graph[current_node]:

            new_distance = distances[current_node] +
graph[current_node][neighbor]

            if new_distance < distances[neighbor]:

                distances[neighbor] = new_distance

                queue.append(neighbor)

    return distances
```

A priority queue is the main element in this function which helps us keep track of distances from sources to other nodes which are present in the graph. The algorithm adds a source node to the priority queue and the priority queue is distinguished by the distances of the nodes in the queue. Shortest distance of node from source node will be present at the front of the queue. The function here eliminates when the priority queue is emptying which tells us nodes present in the graph have been traveled and their distances to the source node have been calculated.

4. **The space complexity of a binary search tree**

The space complexity of a binary search tree is **O(n)**, where n is the number of nodes in the tree. This is because a binary search tree must store each node in the tree.

5. **What's the time complexity of inserting an element into a Doubly linked list?**

```
class
```

```
Node:
def
__init__(self, data):

self.data = data
        self.next = None
        self.prev = None
class
DoublyLinkedList:
def
__init__(self):
        self.head = None
        self.tail = None

def
insert_at_beginning(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

        if self.tail is

None:
            self.tail = new_node
        else:
            self.head.prev = new_node

    def insert_at_end(self, data):
        new_node = Node(data)
        new_node.prev = self.tail
        self.tail = new_node

        if self.head is None:
```

```
            self.head = new_node
        else:
            self.tail.next = new_node

    def insert_at(self, index, data):
        if index == 0:
            self.insert_at_beginning(data)
        elif index == len(self):
            self.insert_at_end(data)
        else:
            new_node = Node(data)
            current_node = self.head
            for i in
range(index - 1):
                current_node = current_node.next

            new_node.next = current_node.next
            current_node.next.prev = new_node
            new_node.prev = current_node
            current_node.next = new_node
```

For this algorithm, we have inserted a new node at the start of the doubly linked list and at the end we've done the same. To get the index we insert a new node for the specified index. For the time complexity at the beginning and end we've inserted **O(1)**

and at index **O(n)** where **n** is signified as the index.

6. **There's a binary tree; check if it is a binary search tree or not**

```
def is_bst(root):
if root is
None:
```

```
        return
True
if root.left is
not
None
and root.left.data > root.data:
        return
False
if root.right is
not
None
and root.right.data < root.data:
        return
False
    return is_bst(root.left) and is_bst(root.right)
```

For the binary search tree check system, we've defined a function at first to check if it's a binary search tree by recursively validating node values.

7. **Given a linked list, determine if it has a cycle**

```
def has_cycle(head):
    slow = head
    fast = head
    while fast is not None and fast.next is not None:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True
    return False
```

For this code what we can do is create a function using the well-known **Floyd's Tortoise** and **Hare algorithm** to check if a linked list has a cycle or not.

8. **There's a string; determine if it is a palindrome or not.**

```
def is_palindrome(string):
    string = string.lower()
    left = 0
    right = len(string) - 1
    while left < right:
        if string[left] != string[right]:
            return False
        left += 1
        right -= 1
    return True
```

Here we have implemented a function is_palindrome to check if the string is a palindrome or not. By using left and right pointers to compare the characters from start and end to the string moving towards the center.

9. **Given an array of integers, find the two numbers that add up to a given target sum.**

```
def find_two_numbers_with_sum(array, target_sum):
    seen = set()
    for number in array:
        complement = target_sum - number
        if complement in seen:
            return (number, complement)
        seen.add(number)
    return None
```

In this function, we've used find_two_numbers_with_sum element to find two numbers in an array that gives us a target addition. It uses a set to track numbers which are present while iterating from the array.

10. **How to find the kth largest element in an array in linear time gives a brief explanation.**

```
def quickselect(array, k):
  if len(array) == 1:
    return array[0]
  pivot = array[random.randint(0, len(array) - 1)]
  less_than_pivot = []
  greater_than_pivot = []
  for element in array:
    if element < pivot:
      less_than_pivot.append(element)
    elif element > pivot:
      greater_than_pivot.append(element)
  if k <= len(less_than_pivot):
    return quickselect(less_than_pivot, k)
  elif k > len(less_than_pivot) + 1:
    return quickselect(greater_than_pivot, k - len(less_than_pivot) - 1)
  else:
    return pivot
```

The above code uses the QuickSelect algorithm which helps us to find the k-th smallest element in an unordered list.

Base Case, Pivot Selection, Partitioning, and Recursion are the four main elements which we use in this function to get the desired results.

11. **How do you check if a string is a palindrome or not? Write a code and explain.**

```
def is_palindrome(string):
```

```
    string = string.lower()

    reversed_string = string[::-1]

    return string == reversed_string
```

Here the algorithm is pretty easy as we've converted the input string to lowercase and then reverse the string and given a command to Returns True if the original string which is added is equal to its reverse, indicating a palindrome.

12. **Write a function to find the factorial of a number.**

```
def factorial(number):

    if number == 0:

        return 1

    return number * factorial(number - 1)
```

For the factorial function we've used recursion to calculate the factorial of the number. The base case here is factorial(0) returns 1 and the recursive case here multiplies the number by the factorial of - 1.

13. **Write a code to find out the greatest common divisor (GCD) of two numbers.**

```
def gcd(a, b):

    while b != 0:

        a, b = b, a % b

    return a
```

To get the Greatest Common Divisor we've used the Euclidean Algorithm mixed with a while loop. The algorithm then Swaps and calculates the remainder until the number at the second position becomes zero. Then it returns the answer

14. **How you reverse a linked list.**

```
def reverse_linked_list(head):
```

```
    if head is None or head.next is None:
        return head
    new_head = None
    while head is not None:
        next = head.next
        head.next = new_head
        new_head = head
        head = next
    return new_head
```

For the reverse linked list we Iteratively reverse a linked list by adjusting its given pointers. We Use three pointers head, new_head, and next in the program to reverse the links.

15. **Write a function to clone a linked list.**

```
def clone_linked_list(head):
    if head is None:
        return None
    new_head = Node(head.data)
    current = head
    new_current = new_head
    while current.next is not None:
        new_node = Node(current.next.data)
        new_current.next = new_node
        new_current = new_node
        current = current.next
    return new_head
```

For a Clone Linked List we Create a new linked list with mixing nodes which have the same data as the original linked list present. Then it Iterates through the original list, making new nodes for each.

Also Read :

Microsoft Explore Internship

Google Internship 2024

16. **Write a function to find the lowest common ancestor of two nodes in a binary tree.**

```
def lowest_common_ancestor(root, p, q):
    if root is None or root == p or root == q:
        return root
    left = lowest_common_ancestor(root.left, p, q)
    right = lowest_common_ancestor(root.right, p, q)
    if left and right:
        return root
    return left or right
```

For this code to get the Lowest Common Ancestor in a Binary Tree we Recursively find the lowest common ancestor of two nodes in a binary tree. Then it returns the root when it comes forward to a node and mixes the results from left and right subtrees.

17. **Write a code and explain the easiest way to check if a binary tree is balanced or not.**

```
def is_balanced(root):
    if root is None:
        return True
    left_height = height(root.left)
    right_height = height(root.right)
    return abs(left_height - right_height) <= 1 and is_balanced(root.left)
and is_balanced(root.right)
```

For the Balanced Binary Tree Check algorithm we first Recursively checks if a binary tree is balanced or not. Then we compare the heights of the left and right subtrees and make sure that their difference is at most 1.

18. **Write a function to find the k largest elements in an array of integers.**

```
def find_k_largest_elements(array, k):
    """
    Args:
        array: A list of integers.
        k: The number of largest elements to find.
    Returns:
        A list of the k largest elements in the array, in descending order.
    """
```

For the largest element in the array a function header is given and it would involve sorting the array and returning the elements.

19. **How you delete a node from a linked list.**

```
def delete_node(head, node):
    """
    Deletes a node from a linked list.

    Args:
        head: The head of the linked list.
        node: The node to delete.

    Returns:
        The head of the linked list after the node has been deleted.
    """
    if node == head:
        head = head.next
        return head
```

```
    previous_node = None
    current_node = head
    while current_node != node:
        previous_node = current_node
        current_node = current_node.next
    previous_node.next = node.next
```

First up if the node which is to be deleted is head then we update the head to the next node and return to the new head. If it's not head we traverse the list until we find the node to delete. The next pointer will be updated of the previous node to skip the node which is to be deleted in the code.

20. **Given a graph, check if it is bipartite.**

```
def is_bipartite(graph):
    colors = {}

    for node in graph:
        if node not in colors:
            colors[node] = 0

        if colors[node] == 1:
            for neighbor in graph[node]:
                if neighbor in colors and colors[neighbor] == 1:
                    return False

        for neighbor in graph[node]:
            if neighbor not in colors:
                colors[neighbor] = 1 - colors[node]
```

```
    return True
```

The color nodes of a graph with two colors is 0 and 1 and adjacent nodes have very different colors. Here we use a dictionary which is of colors to store the color of every node that is present to have different colors. By checking if the coloring is valid by ensuring no adjacent nodes have the same color in the code. Then we Return True if the graph is bipartite and False is not.