Advanced Algorithmic Problem Solving

mid term answers

PRACTICE QUESTIONS FOR ETE

1. Analyse how different hashing methods can be applied to optimize search and sort operations.

Search Optimization with Hashing:

- **Hash Table:** Imagine a data structure like an indexed filing cabinet. Each item has a unique "key" (like an employee ID). A hash function takes that key and calculates an "address" (like a cabinet number) where the item is stored.
- **Fast Lookup:** By calculating the address for the item you're searching for, you can jump directly to that location in the hash table, bypassing the need to scan through the entire dataset. This is significantly faster than linear search, especially for large datasets.

Types of Hashing and Considerations:

- **Collision:** Multiple keys might map to the same address (collision). Techniques like separate chaining (storing multiple items at the same address) or open addressing (probing for the next available address) can handle collisions but add some overhead.
- **Hash Function Choice:** A good hash function should distribute keys uniformly across the available addresses to minimize collisions and optimize search time.

2. Analyse the performance and application scenarios of binomial and Fibonacci heaps.

Heap Type	Insertions/Merges	Other Operations	Applications
Binomial	Faster than Binary Heaps	Slower than Fibonacci Heaps	Frequent merging/large insertions
Fibonacci	Amortized constant time	More complex structure	High-speed insertions & merges (Dijkstra's algorithm)

3. Investigate the disjoint set union data structure and its operations, evaluating its efficiency in different applications.

Disjoint Set Union (DSU) tracks separate groups (sets) of elements. It lets you:

- **Merge groups:** Combine two separate sets into one.
- **Check group membership:** See which group an element belongs to.

Super fast: DSU finds groups and merges them in near-constant time (average) on large datasets, making it very efficient.

Used for:

- Finding minimum spanning trees (electrical wires)
- Grouping friends in a social network
- Modeling connected areas (like fire spread)

Advanced Algorithmic Problem Solving

mid term answers

4. Compare and contrast Depth-First Search (DFS) and Breadth-First Search (BFS) in various contexts.

DFS vs. BFS

Feature	DFS	BFS
Traversal Strategy	Goes deep into one branch before backtracking	Expands outward level by level
Finding Specific Node	Slower for distant nodes, faster for close ones	Faster, especially for nodes near the start
Connectivity Check	Efficient	Less efficient
Cycle Detection	More suitable due to revisiting nodes	Less suitable
Shortest Path (unweighted)	Not guaranteed	Guaranteed
Data Structure	Stack (LIFO)	Queue (FIFO)

5. Evaluate shortest path algorithms, minimum spanning tree algorithms, and their applications in real-world problems.

Shortest path and minimum spanning tree (MST) algorithms are crucial in computer science with significant real-world applications.

Shortest Path Algorithms

1. **Dijkstra's Algorithm**: Finds shortest paths in weighted graphs with non-negative weights. Applications: GPS navigation, network routing, transportation planning.
2. **Bellman-Ford Algorithm**: Handles graphs with negative weights. Applications: Currency arbitrage, telecommunications routing.
3. **A* Algorithm**: Uses heuristics for efficient pathfinding. Applications: Video games, robotic motion planning.

MST Algorithms

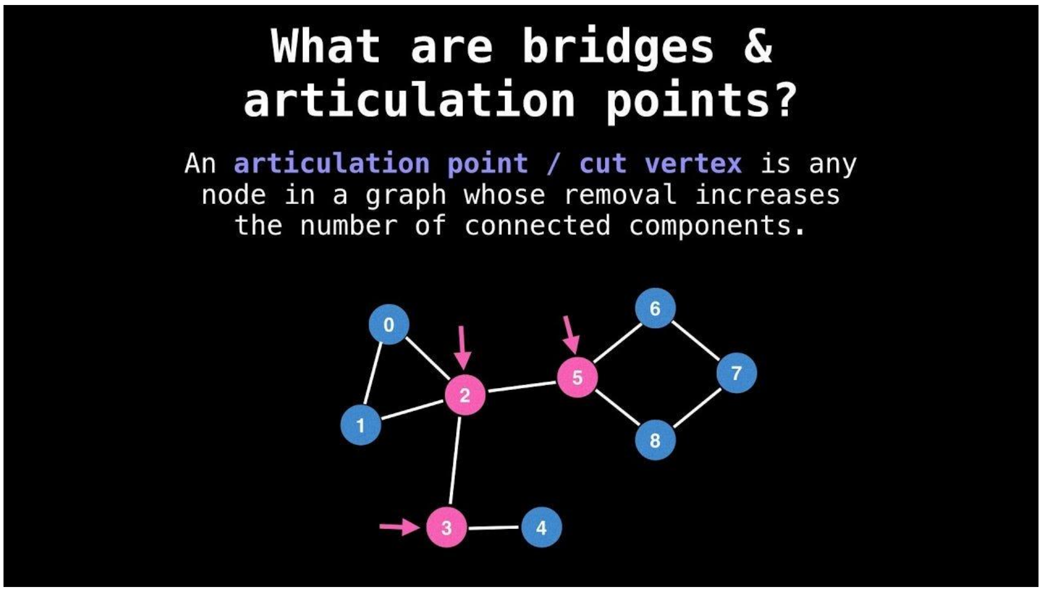
1. **Kruskal's Algorithm**: Builds MST by adding smallest edges without cycles. Applications: Network design, clustering.
2. **Prim's Algorithm**: Grows MST from an arbitrary vertex. Applications: Communication networks, road network optimization.

Applications

1. **Telecommunications**: Optimizing data routing and network costs.
2. **Transportation and Logistics**: Route optimization and infrastructure planning.
3. **Urban Planning**: Traffic flow optimization and utility planning.
4. **Computer Networks**: Efficient data transfer and network topology design.

6 Investigate the significance of articulation points and bridges in network design and reliability.

Advanced Algorithmic Problem Solving
mid term answers



What are bridges & articulation points?

An **articulation point / cut vertex** is any node in a graph whose removal increases the number of connected components.

Significance

1. **Articulation Points**:
   - Critical nodes; their failure disrupts network connectivity.
   - Enhances network robustness.
2. **Bridges**:
   - Critical connections; their failure disconnects parts of the network.
   - Ensures path redundancy.

Applications

1. **Communication Networks**: Maintain connectivity and prevent partitioning.
2. **Transportation Networks**: Protect critical routes and ensure traffic flow.
3. **Utility Networks**: Safeguard service continuity.

7 Examine Strasson's matrix multiplication algorithm and compare it with conventional methods.

Strassen's Matrix Multiplication Algorithm

- **Overview**: Efficient algorithm that reduces matrix multiplication time complexity.
- **Key Idea**: Divides matrices into submatrices and performs only 7 multiplications instead of 8.
- **Time Complexity**: O(nlog⁡27)≈O(n2.81)O(n^{\log_2{7}}) \approx O(n^{2.81})O(nlog27)≈O(n2.81)
- **Best For**: Large matrices where computational efficiency is critical.
- **Drawbacks**: More complex and requires more memory.

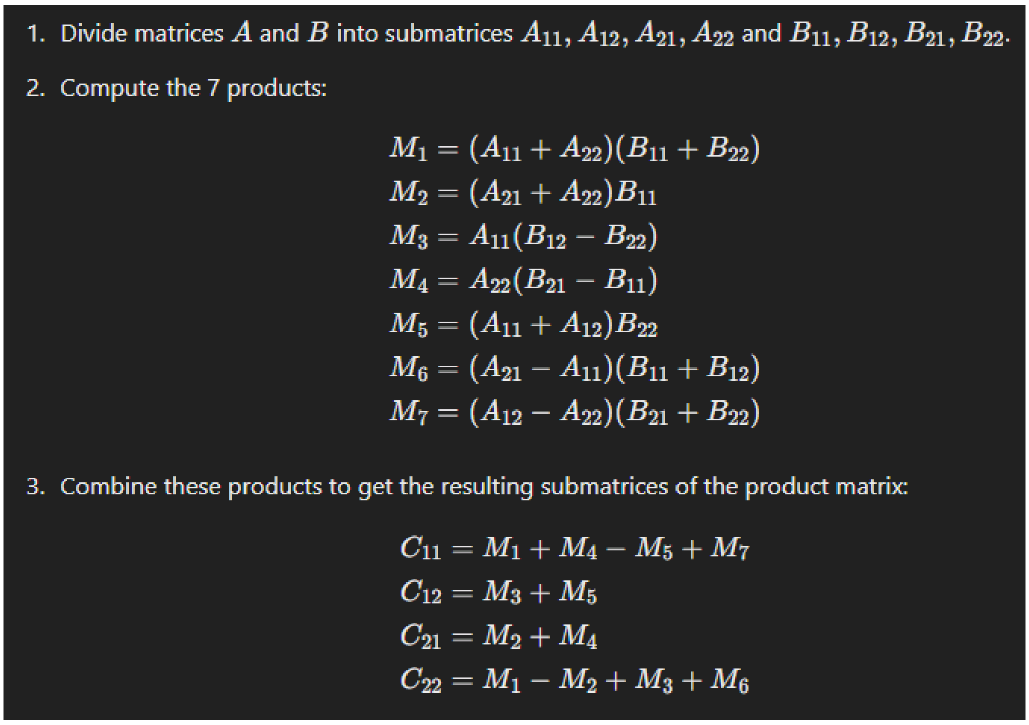
Conventional Matrix Multiplication

- **Overview**: Standard method using three nested loops.
- **Time Complexity**: O(n3)O(n^3)O(n3)
- **Best For**: Small matrices due to simplicity and lower overhead.

- **Advantages**: Simpler implementation, less memory usage.

Comparison

1. **Time Complexity**:
   - **Conventional**: O(n3)O(n^3)O(n3)
   - **Strassen's**: O(n2.81)O(n^{2.81})O(n2.81)
2. **Efficiency**:
   - **Strassen's**: Faster for large matrices.
   - **Conventional**: More practical for small matrices.
3. **Memory Usage**:
   - **Strassen's**: Higher memory usage.
   - **Conventional**: Lower memory usage.
4. **Practical Use**:
   - **Strassen's**: Large-scale applications, theoretical computer science.
   - **Conventional**: Simpler, straightforward for smaller tasks.



1. Divide matrices A and B into submatrices $A_{11}, A_{12}, A_{21}, A_{22}$ and $B_{11}, B_{12}, B_{21}, B_{22}$.
2. Compute the 7 products:

$$\begin{aligned}
M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22})B_{11} \\
M_3 &= A_{11}(B_{12} - B_{22}) \\
M_4 &= A_{22}(B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12})B_{22} \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}$$

3. Combine these products to get the resulting submatrices of the product matrix:

$$\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M_3 + M_6
\end{aligned}$$

8 Analyse the time complexity and efficiency of algorithms based on divide and conquer, such as counting inversions and finding the closest pair of points.

Attribute	**Counting Inversions**	**Closest Pair of Points**
Algorithm	Counting Inversions	Closest Pair of Points
Problem	Counting inversions in an array	Finding the closest pair of points

Key Idea	Divide the array, count inversions, merge results	Divide points, find closest pairs, merge results
Time Complexity	O(nlog⁡n)O(n \log n)O(nlogn)	O(nlog⁡n)O(n \log n)O(nlogn)
Efficiency	Efficient for large arrays	Efficient for large point sets
Best Use	Sorting-related tasks, data analysis, algorithmic competitions	Geometric problems, clustering, computational geometry

9 Critically evaluate the effectiveness of universal hashing in various scenarios.

Universal hashing is a technique that improves the performance of hash functions by selecting one at random from a family of hash functions, minimizing the chance of collisions. Here's a summary of its effectiveness in various scenarios:

1. **General-purpose Hashing**: Highly effective for unknown input distributions, reducing worst-case collisions.
2. **Cryptographic Applications**: Not suitable due to lack of required security properties.
3. **Hash Tables and Data Structures**: Improves performance and reduces collisions, ensuring consistent O(1) operations.
4. **Adversarial Inputs**: Effective against collision attacks by making it hard to predict the hash function.
5. **Real-time Systems**: Improves performance predictability but may introduce slight delays due to random selection.
6. **Distributed Systems**: Enhances load balancing by ensuring even key distribution across nodes.
7. **Memory-limited Environments**: Additional memory and computational overhead may be a drawback.

10 Judge the suitability of hashing methods for optimization problems and justify the chosen method.

- **Simple Hashing**
  - **Suitability**: Small datasets, predictable distributions.
  - **Pros**: Easy to implement, fast.
  - **Cons**: Prone to collisions with larger or unpredictable datasets.
- **Universal Hashing**
  - **Suitability**: Unknown or adversarial input distributions.

- **Pros**: Reduces collision probability, robust against worst-case scenarios.
- **Cons**: Slightly more complex due to random selection of hash functions.

- **Perfect Hashing**
  - **Suitability**: Static datasets with known keys.
  - **Pros**: O(1) lookup time without collisions.
  - **Cons**: Only for static data; not suitable for dynamic datasets.

- **Cuckoo Hashing**
  - **Suitability**: High-performance insertions and lookups, real-time systems.
  - **Pros**: Guaranteed O(1) lookup time, handles high load factors.
  - **Cons**: More complex to implement.

- **Consistent Hashing**
  - **Suitability**: Distributed systems, load balancing.
  - **Pros**: Minimizes remapping during resizing, balanced load distribution.
  - **Cons**: Slightly more complex.

- **Dynamic Perfect Hashing**
  - **Suitability**: Changing datasets over time.
  - **Pros**: Efficient operations, adapts to dataset changes.
  - **Cons**: Complex to implement.

11 Judge the effectiveness of shortest path algorithms and minimum spanning tree algorithms in solving real-world problems.

Shortest Path Algorithms

1. **Dijkstra's Algorithm**
   - **Pros**: Efficient for non-negative weighted graphs, used in network routing.
   - **Cons**: Ineffective with negative weights, slower for large graphs.
2. **Bellman-Ford Algorithm**
   - **Pros**: Handles negative weights, detects negative cycles, used in financial modeling.
   - **Cons**: Slower than Dijkstra's for non-negative weights.
3. *A Algorithm**
   - **Pros**: Fast with good heuristics, used in AI and robotics for pathfinding.
   - **Cons**: Heuristic-dependent, unsuitable for negative weights.
4. **Floyd-Warshall Algorithm**
   - **Pros**: Finds shortest paths for all pairs in dense graphs, simple implementation.
   - **Cons**: High time complexity, impractical for large graphs.

Minimum Spanning Tree (MST) Algorithms

Advanced Algorithmic Problem Solving
mid term answers

1. **Kruskal's Algorithm**
   - **Pros**: Simple, effective for sparse graphs, used in network design.
   - **Cons**: Sorting edges can be time-consuming for large graphs.
2. **Prim's Algorithm**
   - **Pros**: Efficient for dense graphs, used with priority queues in network construction.
   - **Cons**: Less intuitive in some applications.
3. **Borůvka's Algorithm**
   - **Pros**: Suitable for parallel processing, handles large graphs.
   - **Cons**: Complex implementation.

Real-World Applications

- **Shortest Path Algorithms**: Used in network routing, GPS navigation, and transportation logistics for optimal route planning.
- **Minimum Spanning Tree Algorithms**: Applied in network design, data clustering, and electrical grid infrastructure to minimize costs.

11 Justify the choice of graph algorithms based on problem constraints and requirements.

Shortest Path Algorithms

1. **Dijkstra's Algorithm**
   - **Use When**: You have a graph with non-negative weights.
   - **Why**: It's efficient (O(V^2) or O(E + V log V) with a priority queue) and straightforward for such graphs.
   - **Example**: Finding the shortest route in a city map without negative distances.
2. **Bellman-Ford Algorithm**
   - **Use When**: You need to handle negative weights or detect negative cycles.
   - **Why**: It works with negative weights and can identify negative cycles, though it’s slower (O(VE)).
   - **Example**: Financial models where costs can decrease.
3. *A Algorithm**
   - **Use When**: You need fast pathfinding with a known heuristic.
   - **Why**: It uses heuristics to speed up the search, making it efficient in practice.
   - **Example**: GPS navigation systems considering real-time traffic.
4. **Floyd-Warshall Algorithm**
   - **Use When**: You need shortest paths between all pairs of nodes in a dense graph.
   - **Why**: It’s simple and handles all pairs but has high time complexity (O(V^3)).
   - **Example**: Analyzing all possible routes in a small, densely connected network.

Minimum Spanning Tree (MST) Algorithms

1. **Kruskal's Algorithm**

- **Use When**: You have a sparse graph and can sort edges efficiently.
- **Why**: It's simple and works well for sparse graphs.
- **Example**: Designing a cost-effective network with fewer connections.

2. **Prim's Algorithm**
   - **Use When**: You have a dense graph or prefer adjacency lists.
   - **Why**: It's efficient for dense graphs and works well with priority queues.
   - **Example**: Building a network with many connections.
3. **Borůvka's Algorithm**
   - **Use When**: You can use parallel processing and need to handle very large graphs.
   - **Why**: It's suitable for parallel execution and large datasets.
   - **Example**: Large-scale network design requiring parallel computation.

Summary

- **Dijkstra** for non-negative weights.
- **Bellman-Ford** for negative weights.
- **A*** for fast pathfinding with heuristics.
- **Floyd-Warshall** for all-pairs shortest paths in dense graphs.
- **Kruskal** for sparse MSTs.
- **Prim** for dense MSTs.
- **Borůvka** for parallel processing and large graphs.

12 Create a system that dynamically selects the appropriate search method based on the dataset characteristics.

1. **Input Analysis**:
   - **Graph Type**: Determine if the graph is dense or sparse.
   - **Edge Weights**: Check if the graph contains negative weights.
   - **Path Requirements**: Identify if you need a single-source shortest path or all-pairs shortest paths.
2. **Decision Criteria**:
   - **Non-negative Weights**:
     - **Single Source**: Use Dijkstra's algorithm.
     - **All Pairs**: Use Floyd-Warshall if the graph is dense.
   - **Negative Weights**:
     - **Single Source**: Use Bellman-Ford.
   - **Heuristic-Based Search**: Use A* if a good heuristic is available.
   - **Minimum Spanning Tree (MST)**:
     - **Sparse Graph**: Use Kruskal's algorithm.
     - **Dense Graph**: Use Prim's algorithm.
     - **Parallel Processing**: Use Borůvka's algorithm.
3. **Dynamic Selection System** (Pseudocode):

```
python
Copy code
def select_algorithm(graph, heuristic=None):
    if is_mst_problem(graph):
        if is_dense(graph):
            return "Prim's Algorithm"
        elif is_parallel_processing_available():
            return "Borůvka's Algorithm"
        else:
            return "Kruskal's Algorithm"
    elif is_shortest_path_problem(graph):
        if has_negative_weights(graph):
            return "Bellman-Ford Algorithm"
        elif heuristic:
            return "A* Algorithm"
        elif is_all_pairs_required(graph):
            return "Floyd-Warshall Algorithm" if is_dense(graph)
else "Johnson's Algorithm"
        else:
            return "Dijkstra's Algorithm"
    else:
        return "Algorithm not determined"

# Example Usage
selected_algorithm = select_algorithm(graph,
heuristic=some_heuristic)
print(f"Selected algorithm: {selected_algorithm}")
```

12 Design a robust hashing system that minimizes collisions and optimizes search and sort operations.

Designing a robust hashing system involves several key elements:

1. **Hash Function Selection**: Use a strong hash function like SHA-256 for good distribution of hash values.
2. **Collision Handling**: Implement techniques like chaining or open addressing to manage collisions efficiently.
3. **Load Balancing**: Use consistent hashing and dynamic resizing to balance the load on the hash table.
4. **Indexing and Sorting**: Employ hash indices or hash trees for fast search operations, and use sorting algorithms for sorting hash table entries.
5. **Parallelism and Concurrency**: Support concurrent access with locking mechanisms or lock-free data structures.
6. **Memory Management**: Optimize memory usage and consider caching techniques for performance improvement.
7. **Error Handling and Resilience**: Implement robust error handling and monitoring to ensure system integrity.
8. **Testing and Benchmarking**: Conduct thorough testing and benchmarking to evaluate performance metrics and identify areas for optimization.

13 Design greedy algorithms tailored to solve specific optimization problems like activity selection and task scheduling.

Greedy algorithms provide efficient solutions to specific optimization problems such as activity selection and task scheduling.

- **Activity Selection**:

  - Greedily selecting activities based on their finish times maximizes the number of non-overlapping activities.
  - The algorithm sorts activities by finish time and selects those that do not overlap with previously selected ones.
- **Task Scheduling**:
  - Maximizing total profit by scheduling tasks within deadlines involves sorting tasks by profit and greedily selecting tasks for the latest available slots before their deadlines.
  - The algorithm iterates through tasks, scheduling them in available slots if they do not conflict with previously scheduled tasks.

Greedy algorithms make locally optimal choices at each step, leading to efficient solutions. However, they may not always yield globally optimal results and require careful consideration of problem constraints to ensure desired optimization.

14 Apply the binary search technique to find the first occurrence of a number in a sorted array.

binary search technique to find the first occurrence of a number in a sorted array:

```
python
Copy code
def binary_search_first_occurrence(arr, target):
    low = 0
    high = len(arr) - 1
    result = -1  # Initialize result to -1 (not found)

    while low <= high:
        mid = low + (high - low) // 2

        if arr[mid] == target:
            result = mid  # Update result to current index
            high = mid - 1  # Search for first occurrence on the
left side
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return result

# Example usage
arr = [1, 2, 2, 2, 3, 4, 5]
target = 2
first_occurrence = binary_search_first_occurrence(arr, target)
print("First occurrence index:", first_occurrence)
```

15 Apply the greedy technique to solve the activity selection problem.

Advanced Algorithmic Problem Solving
mid term answers

```
def activity_selection(start_times, finish_times):
    n = len(start_times)
    activities = []  # List to store selected activities
    activities.append(0)  # Add the first activity (assuming sorted by finish times)
    prev_finish = finish_times[0]

    for i in range(1, n):
        if start_times[i] >= prev_finish:
            activities.append(i)
            prev_finish = finish_times[i]

    return activities

# Example usage
start_times = [1, 3, 0, 5, 3, 5, 6, 8, 8, 2, 12]
finish_times = [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
selected_activities = activity_selection(start_times, finish_times)
print("Selected activities:", selected_activities)
```

16 How would you apply stack operations to evaluate a postfix expression? Write the function in C/C++/Java/Python.

```
def evaluate_postfix(expression):
    stack = []

    # Function to perform arithmetic operations
    def apply_operator(op, operand1, operand2):
        if op == '+':
            return operand1 + operand2
        elif op == '-':
            return operand1 - operand2
        elif op == '*':
            return operand1 * operand2
        elif op == '/':
            return operand1 / operand2  # Assuming division is floating-point

    for token in expression.split():
        if token.isdigit():
            stack.append(int(token))
        else:
            operand2 = stack.pop()
            operand1 = stack.pop()
            result = apply_operator(token, operand1, operand2)
            stack.append(result)

    return stack.pop()

# Example usage
postfix_expression = "3 4 + 2 *"
```

result = evaluate_postfix(postfix_expression)
print("Result:", result)

17 Apply binary search tree operations to insert and find an element. Write the function in C/C++/Java/Python.

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        self.root = self._insert_recursive(self.root, key)

    def _insert_recursive(self, root, key):
        if root is None:
            return Node(key)

        if key < root.key:
            root.left = self._insert_recursive(root.left, key)
        elif key > root.key:
            root.right = self._insert_recursive(root.right, key)

        return root

    def find(self, key):
        return self._find_recursive(self.root, key)

    def _find_recursive(self, root, key):
        if root is None or root.key == key:
            return root

        if key < root.key:
            return self._find_recursive(root.left, key)
        else:
            return self._find_recursive(root.right, key)

# Example usage
bst = BinarySearchTree()
bst.insert(50)
bst.insert(30)
bst.insert(20)
bst.insert(40)
bst.insert(70)
bst.insert(60)
bst.insert(80)

found_node = bst.find(40)
if found_node:
    print("Element found:", found_node.key)
else:
    print("Element not found")
```

Advanced Algorithmic Problem Solving
mid term answers

18 Use BFS to implement a level order traversal of a binary tree. Write the function in C/C++/Java/Python.

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def level_order_traversal(root):
    if root is None:
        return []

    result = []
    queue = [root]

    while queue:
        level = []
        level_size = len(queue)

        for _ in range(level_size):
            node = queue.pop(0)
            level.append(node.val)

            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        result.append(level)

    return result

# Example usage
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)

print("Level Order Traversal:", level_order_traversal(root))
```

19 Write a program that uses divide and conquer to find the closest pair of points in a 2D plane.

Advanced Algorithmic Problem Solving

mid term answers

```
import math

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def distance(point1, point2):
    return math.sqrt((point1.x - point2.x) ** 2 + (point1.y - point2.y) ** 2)

def brute_force_closest_pair(points):
    min_distance = float('inf')
    closest_pair = None

    for i in range(len(points)):
        for j in range(i + 1, len(points)):
            dist = distance(points[i], points[j])
            if dist < min_distance:
                min_distance = dist
                closest_pair = (points[i], points[j])

    return min_distance, closest_pair

def closest_pair_divide_conquer(points):
    sorted_points = sorted(points, key=lambda point: point.x)
    return closest_pair_recursive(sorted_points)

def closest_pair_recursive(sorted_points):
    n = len(sorted_points)

    if n <= 3:
        return brute_force_closest_pair(sorted_points)

    mid = n // 2
    left_closest, left_pair = closest_pair_recursive(sorted_points[:mid])
    right_closest, right_pair = closest_pair_recursive(sorted_points[mid:])

    min_dist = min(left_closest, right_closest)

    # Check for points crossing the midpoint
    strip = []
    for point in sorted_points:
        if abs(point.x - sorted_points[mid].x) < min_dist:
            strip.append(point)

    strip_closest, strip_pair = closest_pair_strip(strip, min_dist)

    if strip_closest < min_dist:
        return strip_closest, strip_pair
    elif left_closest < right_closest:
        return left_closest, left_pair
    else:
        return right_closest, right_pair

def closest_pair_strip(strip, min_dist):
    min_distance = min_dist
    closest_pair = None

    strip.sort(key=lambda point: point.y)

    for i in range(len(strip)):
        for j in range(i + 1, len(strip)):
            if strip[j].y - strip[i].y < min_distance:
                dist = distance(strip[i], strip[j])
                if dist < min_distance:
                    min_distance = dist
                    closest_pair = (strip[i], strip[j])
            else:
                break

    return min_distance, closest_pair

# Example usage
points = [Point(2, 3), Point(12, 30), Point(40, 50), Point(5, 1), Point(12, 10),
Point(3, 4)]
closest_distance, closest_pair = closest_pair_divide_conquer(points)
print("Closest Distance:", closest_distance)
print("Closest Pair:", closest_pair)
```

20 Write a program to implement dynamic programming to solve the 0/1 knapsack problem and analyse the memory usage.

```
def knapsack_01(values, weights, capacity):
    n = len(values)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i -
1][w])
            else:
                dp[i][w] = dp[i - 1][w]

    return dp[n][capacity]

# Example usage and memory analysis
values = [60, 100, 120]
weights = [10, 20, 30]
capacity = 50
```

Advanced Algorithmic Problem Solving
mid term answers

```
max_value = knapsack_01(values, weights, capacity)
print("Maximum value in knapsack:", max_value)

# Memory analysis
import sys
size_of_dp = sys.getsizeof(dp)
print("Size of DP table:", size_of_dp, "bytes")
```

21 Evaluate the efficiency of using a sliding window technique for a given dataset of temperature readings over brute force methods.

sliding window technique with brute force methods for analyzing temperature readings:

Aspect	Sliding Window Technique	Brute Force Methods
Time Complexity	O(n) or O(n log n)	O(n^2) or higher
Space Complexity	O(1) or O(n)	Higher than O(n)
Practical Efficiency	Efficient for large datasets and real-time processing	Less efficient for large datasets, more suitable for smaller datasets
Usage	Widely used in real-time data processing and window-based analytics	Used in scenarios with manageable problem sizes or smaller datasets

22 Given a number n, find sum of first *n* natural numbers. To calculate the sum, we will use a recursive function recur_sum().

```
def recur_sum(n):
    if n <= 0:
        return 0
    else:
        return n + recur_sum(n - 1)

# Example usage
n = 5
sum_of_natural_numbers = recur_sum(n)
print("Sum of first", n, "natural numbers:", sum_of_natural_numbers)
```

23 Implement a recursive algorithm to solve the Tower of Hanoi problem. Find its complexity also.

```
def tower_of_hanoi(n, source, target, auxiliary):
    if n == 1:
        print(f"Move disk 1 from {source} to {target}")
        return
    tower_of_hanoi(n-1, source, auxiliary, target)
    print(f"Move disk {n} from {source} to {target}")
    tower_of_hanoi(n-1, auxiliary, target, source)

# Example usage
n = 3
tower_of_hanoi(n, 'A', 'C', 'B')
```

he time complexity of the Tower of Hanoi problem using this recursive algorithm is O(2^n)

24 Given a Binary Search Tree and a node value X, find if the node with value X is present in the BST or not.

```
class TreeNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

def search_bst(root, x):
    if root is None or root.key == x:
        return root is not None

    if x < root.key:
        return search_bst(root.left, x)
    else:
        return search_bst(root.right, x)

# Example usage
root = TreeNode(50)
root.left = TreeNode(30)
root.right = TreeNode(70)
root.left.left = TreeNode(20)
root.left.right = TreeNode(40)
root.right.left = TreeNode(60)
root.right.right = TreeNode(80)

x = 40
if search_bst(root, x):
    print(f"Node with value {x} is present in the BST.")
else:
    print(f"Node with value {x} is not present in the BST.")
```

25 Given two Binary Search Trees. Find the nodes that are common in both of them, ie- find the intersection of the two BSTs.

```
class TreeNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

def in_order_traversal(root, result):
    if root is not None:
        in_order_traversal(root.left, result)
        result.append(root.key)
        in_order_traversal(root.right, result)

def find_common_nodes(root1, root2):
    common_nodes = []
    inorder_list1 = []
    inorder_list2 = []

    in_order_traversal(root1, inorder_list1)
    in_order_traversal(root2, inorder_list2)

    i, j = 0, 0
    while i < len(inorder_list1) and j < len(inorder_list2):
        if inorder_list1[i] == inorder_list2[j]:
            common_nodes.append(inorder_list1[i])
            i += 1
            j += 1
        elif inorder_list1[i] < inorder_list2[j]:
            i += 1
        else:
            j += 1

    return common_nodes

# Example usage
root1 = TreeNode(50)
root1.left = TreeNode(30)
root1.right = TreeNode(70)
root1.left.left = TreeNode(20)
root1.left.right = TreeNode(40)
root1.right.left = TreeNode(60)
root1.right.right = TreeNode(80)

root2 = TreeNode(50)
root2.left = TreeNode(30)
root2.right = TreeNode(70)
root2.left.left = TreeNode(20)
```

```
root2.left.right = TreeNode(40)
root2.right.left = TreeNode(60)
root2.right.right = TreeNode(90)  # Different from root1

common_nodes = find_common_nodes(root1, root2)
print("Common nodes in both BSTs:", common_nodes)
```

26 Design and implement a version of quicksort that randomly chooses pivot elements. Calculate the time and space complexity of algorithm.

```
import random

def randomized_quicksort(arr):
    if len(arr) <= 1:
        return arr

    pivot = random.choice(arr)
    left = [x for x in arr if x < pivot]
    equal = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    return randomized_quicksort(left) + equal + randomized_quicksort(right)

# Example usage
arr = [3, 6, 8, 10, 1, 2, 1]
sorted_arr = randomized_quicksort(arr)
print("Sorted array:", sorted_arr)
```

27 Design and implement an efficient algorithm to merge k sorted arrays.

```
import heapq

def merge_k_sorted_arrays(arrays):
    result = []
    heap = [(array[0], i, 0) for i, array in enumerate(arrays) if array]  # (value,
array_index, element_index)
    heapq.heapify(heap)

    while heap:
        val, arr_idx, elem_idx = heapq.heappop(heap)
        result.append(val)

        if elem_idx + 1 < len(arrays[arr_idx]):
            next_val = arrays[arr_idx][elem_idx + 1]
            heapq.heappush(heap, (next_val, arr_idx, elem_idx + 1))

    return result

# Example usage
arrays = [[1, 3, 5], [2, 4, 6], [0, 7, 8, 9]]
merged_array = merge_k_sorted_arrays(arrays)
print("Merged sorted array:", merged_array)
```

28 Given two strings str1 & str 2 of length n & m respectively, find the length of the longest subsequence present in both. A subsequence is a sequence that can be derived from the given string by deleting some or no elements without changing the order of the remaining elements. For example, "abe" is a subsequence of "abcde". Example :Input: n = 6, str1 = ABCDGH and m = 6, str2 = AEDFHR Output: 3 Explanation: LCS for input strings "ABCDGH" and "AEDFHR" is "ADH" of length 3.

```
def longest_common_subsequence(str1, str2):
    n = len(str1)
    m = len(str2)
    dp = [[0] * (m + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if str1[i - 1] == str2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    return dp[n][m]

# Example usage
str1 = "ABCDGH"
str2 = "AEDFHR"
result = longest_common_subsequence(str1, str2)
print("Length of longest common subsequence:", result)
```

For the given example input, the output will be 3, which is the length of the longest common subsequence "ADH" between "ABCDGH" and "AEDFHR".

29 You are given an amount denoted by **value**. You are also given an array of coins. The **array** contains the **denominations** of the given coins. You need to find the **minimum number of coins** to make the change for **value** using the coins of given denominations. Also, keep in mind that you have **infinite supply** of the coins. Input:
value = 10
numberOfCoins = 4
coins[] = {2 5 3 6}
Output: 2

```
def min_coins(value, coins):
    n = len(coins)
    dp = [float('inf')] * (value + 1)
    dp[0] = 0

    for i in range(1, value + 1):
        for j in range(n):
            if coins[j] <= i:
                dp[i] = min(dp[i], dp[i - coins[j]] + 1)

    return dp[value]

# Example usage
value = 10
coins = [2, 5, 3, 6]
result = min_coins(value, coins)
print("Minimum number of coins required:", result)
```

30 Hashing is very useful to keep track of the frequency of the elements in a list. You are given an array of integers. You need to print the count of non-repeated elements in the array. Example : Input:1 1 2 2 3 3 4 5 6 7 Output:4

```
def count_non_repeated_elements(arr):
    frequency = {}
    non_repeated_count = 0

    for num in arr:
        frequency[num] = frequency.get(num, 0) + 1

    for key, value in frequency.items():
        if value == 1:
            non_repeated_count += 1

    return non_repeated_count

# Example usage
arr = [1, 1, 2, 2, 3, 3, 4, 5, 6, 7]
result = count_non_repeated_elements(arr)
print("Count of non-repeated elements:", result)
```

31 Given two arrays a[] and b[] of size n and m respectively. The task is to find the number of elements in the union between these two arrays. Union of the two arrays can be defined as the set containing distinct elements from both the arrays. If there are repetitions, then only one occurrence of element should be printed in the union. Input:1 2 3 4 5 1 2 3 Output: 5

```
def count_union_elements(arr1, arr2):
    union_set = set(arr1) | set(arr2)  # '|' operator computes the union of sets
    return len(union_set)

# Example usage
arr1 = [1, 2, 3, 4, 5]
arr2 = [1, 2, 3]
result = count_union_elements(arr1, arr2)
print("Number of elements in the union:", result)
```

32 Inorder traversal means traversing through the tree in a Left, Node, Right manner. We first traverse left, then print the current node, and then traverse right. This is done recursively for each node. Given a BST, find its in-order traversal.

```
class TreeNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

def inorder_traversal(root):
    result = []
    if root:
        result.extend(inorder_traversal(root.left))
        result.append(root.key)
        result.extend(inorder_traversal(root.right))
    return result

# Example usage
root = TreeNode(5)
root.left = TreeNode(3)
root.right = TreeNode(8)
root.left.left = TreeNode(2)
root.left.right = TreeNode(4)
root.right.left = TreeNode(6)
root.right.right = TreeNode(9)

inorder_result = inorder_traversal(root)
print("In-order traversal:", inorder_result)
```

For More Click

Advanced Algorithmic Problem Solving(R1UC601B)

PRACTICE QUESTIONS FOR MTE

1. What is meant by time complexity and space complexity? Explain in detail.

Time Complexity and **Space Complexity** are two important aspects of evaluating the efficiency of an algorithm.

Time Complexity

Time Complexity is a measure of the amount of time an algorithm takes to run as a function of the size of the input to the program. It's usually expressed using Big O notation, which describes the upper bound of the time complexity in the worst-case scenario.

For example, for a simple for loop iterating over n elements, the time complexity would be O(n). This means that if the size of the input (n) doubles, the time taken for the algorithm also doubles.

Space Complexity

Space Complexity is a measure of the amount of memory an algorithm needs to run to completion. It's also expressed using Big O notation, which describes the upper bound of the space complexity in the worst-case scenario.

For instance, consider an algorithm that creates a new list and adds n elements to it. The space complexity for this algorithm is O(n). This means if the size of the input (n) doubles, the memory required by the algorithm also doubles.

In summary, time complexity and space complexity help us to estimate the worst-case scenario in terms of time and space respectively, for an algorithm. They are crucial in determining the scalability and efficiency of an algorithm. However, there's often a trade-off between the two. For example, an algorithm might be able to run faster (lower time complexity) by using more memory (higher space complexity), and vice versa. This trade-off is often referred to as the time-space trade-off.

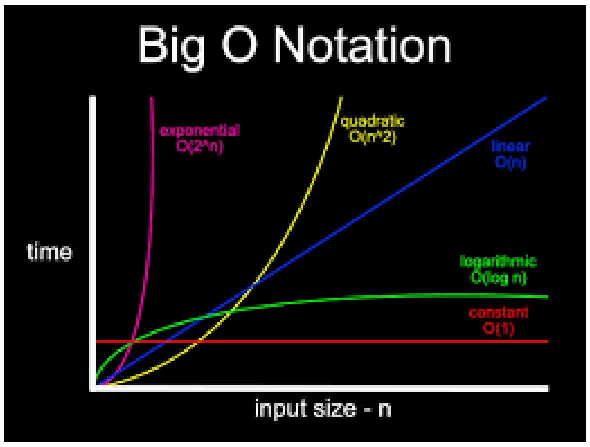
2. What are asymptotic notations? Define Theta, Omega, big O, small omega, and small o.

Asymptotic notations are mathematical tools used to describe the limiting behavior of functions when the argument tends towards a particular value or infinity, most often in terms of simpler functions. They are widely used in computer science to describe the time and space complexity of algorithms. Here are the definitions of the notations you asked about:

1. **Big O Notation (O)**: The Big O notation upper bounds a function to within a constant factor. It is used to describe the worst-case scenario of an algorithm's time or space complexity. For example, if we say a function f(n) is O(g(n)), it means there are some constant c and some value of n (n0) after which f(n) is always less than or equal to c*g(n).
2. **Omega Notation (Ω)**: The Omega notation provides an asymptotic lower bound. It is used to describe the best-case scenario of an algorithm's time or space complexity. For example, if we say a function f(n) is Ω(g(n)), it means there are some constant c and some value of n (n0) after which f(n) is always greater than or equal to c*g(n).
3. **Theta Notation (Θ)**: The Theta notation bounds a function from above and below, so it defines exact asymptotic behavior. A function f(n) has a Θ(g(n)) if it is both O(g(n)) and Ω(g(n)).
4. **Small o notation (o)**: The small o notation is an upper bound that is not tight. If f(n) is o(g(n)), then for any constant c > 0, there exists a constant n0 such that 0 ≤ f(n) < c*g(n) for all n > n0. This means g(n) grows strictly faster than f(n).
5. **Small omega notation (ω)**: The small omega notation is a lower bound that is not tight. If f(n) is ω(g(n)), then for any constant c > 0, there exists a constant n0 such that 0 ≤ c*g(n) < f(n) for all n > n0. This means f(n) grows strictly faster than g(n).

3. Explain the meaning of O(2^n), O(n^2), O(nlgn), O(lg n). Give one example of each.

1. **O(2^n)**: This represents an algorithm whose growth doubles with each addition to the input data set. The growth curve of an O(2^n) function is exponential - starting off very shallow, then rising meteorically. An example of an O(2^n) function is the recursive calculation of Fibonacci numbers.
2. **O(n^2)**: This time complexity represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This is common with algorithms that involve nested iterations over the data set. An example is bubble sort.
3. **O(n log n)**: This is the time complexity of an algorithm that increases linearly with the size of the input, but also has a logarithmic aspect due to some divide and conquer aspect of the algorithm. Common algorithms with this time complexity include quicksort, mergesort, and heapsort.
4. **O(log n)**: Logarithmic complexity arises when we reduce the size of the input data in each step of the algorithm (as in binary search). Here's a simple binary search algorithm:



4. Explain sliding window protocol.

The **Sliding Window Technique** is a method used in Data Structures and Algorithms (DSA). It's used to efficiently solve problems that involve defining a window or range in the input data (arrays or strings) and then moving that window across the data to perform some operation within the window. This technique is commonly used in algorithms like finding subarrays with a specific sum, finding the longest substring with unique characters, or solving problems that require a fixed-size window to process elements efficiently.

Here's a simple example to illustrate this: say we have an array of size N and also an integer K. Now, we have to calculate the maximum sum of a subarray having size exactly K. One way to do this is by taking each subarray of size K from the array and find out the maximum sum of these subarrays. This can be done using Nested loops which will result into O (N^2) Time Complexity.

But we can optimize this approach using the Sliding Window Technique. Instead of taking each K sized subarray and calculating its sum, we can just take one K size subarray from 0 to K-1 index and calculate its sum. Now shift our range one by one along with the iterations and update the result. This operation is optimal because it takes O (1) time to shift the range instead of recalculating.

There are basically two types of sliding window:

1. **Fixed Size Sliding Window**: Compute the result for the 1st window, i.e., include the first K elements of the data structure. Then use a loop to slide the window by 1 and keep computing the result window by window.
2. **Variable Size Sliding Window**: The steps to solve these questions are similar to the fixed size sliding window, but the window size can change based on certain conditions.

5. Explain Naive String-Matching algorithm. Discuss its time and space complexity.

The Naive String-Matching Algorithm is a simple method used for pattern searching within a text. Given a text string of length n and a pattern of length m, the task is to print all occurrences of the pattern in the text. You may assume that n > m.

Here's how the algorithm works:

1. Slide the pattern over the text one by one.
2. For each character in the text string, check for a pattern match.
3. If a match is found, then slide by 1 again to check for subsequent matches.

Now, let's discuss its time and space complexity:

Time Complexity:

- **Best Case:** O(n) - This is when the pattern is found at the very beginning of the text (or very early on). The algorithm will perform a constant number of comparisons, typically on the order of O(n) comparisons, where n is the length of the pattern.
- **Worst Case:** O(n*m) - This is when the pattern doesn't appear in the text at all or appears only at the very end. The algorithm will perform O((n-m+1)*m) comparisons, where n is the length of the text and m is the length of the pattern. In the worst case, for each position in the text, the algorithm may need to compare the entire pattern against the text.

Space Complexity:

- The space complexity of the Naive String-Matching Algorithm is O(1). This is because it does not require any additional space that scales with input size.

```
def naive_string_matching(text, pattern):
  n = len(text)
  m = len(pattern)

  # Slide the pattern over the text one by one
  for i in range(n - m + 1):
    j = 0

    # For current index i, check for pattern match
    while(j < m):
      if (text[i + j] != pattern[j]):
        break
      j += 1

    # If pattern matches at index i
    if (j == m):
      print("Pattern found at index", i)

# Driver's Code
text = "AABAACAADAABAABA"
pattern = "AABA"
naive_string_matching(text, pattern)
```

6 Explain Rabin Karp String-Matching algorithm. Discuss its time and space complexity.

The Rabin-Karp Algorithm is a pattern searching algorithm that uses hashing to find any one of a set of pattern strings in a text. It uses a rolling hash to quickly filter out positions of the text that cannot match the pattern. This is why it's faster than the Naive String-Matching Algorithm for most cases.

Here’s how the algorithm works:

1. Calculate the hash value of the pattern.
2. Calculate the hash value of the first window of the text of the same length as the pattern.
3. Compare the hash value of the pattern with the hash value of the current substring of text.
4. If the hash values match, then only it starts matching individual characters.
5. Slide the window in the text one character at a time, updating the hash value of the text window at each step.

The hash value is calculated using a rolling hash function, which allows you to update the hash value for a new substring by efficiently removing the contribution of the old character and adding the contribution of the new character. This makes it possible to slide the pattern over the text and calculate the hash value for each substring without recalculating the entire hash from scratch.

Now, let’s discuss its time and space complexity:

Time Complexity:

- **Best Case:** O(n + m) - This is when the pattern is found at the very beginning of the text (or very early on). The algorithm will perform a constant number of comparisons, typically on the order of O(n + m) comparisons, where n is the length of the text and m is the length of the pattern.
- **Worst Case:** O(nm) - This is when all characters of pattern and text are the same as the hash values of all the substrings of T[] match with the hash value of P[]. The algorithm will perform O((n-m+1)*m) comparisons, where n is the length of the text and m is the length of the pattern[1].

Space Complexity:

- The space complexity of the Rabin-Karp Algorithm is O(1). This is because it does not require any additional space that scales with input size[2].

```
function RabinKarp(string s[1..n], string pattern[1..m])
  hpattern := hash(pattern[1..m]);  hs := hash(s[1..m])
  for i from 1 to n-m+1
    if hs = hpattern
      if s[i..i+m-1] = pattern[1..m]
        return i
    hs := hash(s[i+1..i+m])
  return not found
```

7 Explain Knuth Morris and Pratt String-Matching algorithm. Discuss its time and space complexity.

It’s a linear time algorithm used for pattern searching in strings.

The KMP algorithm uses the observation that when a mismatch occurs, the pattern itself contains enough information to determine where the next match could begin. This avoids unnecessary comparisons in the future.

Here's a high-level overview of how it works:

1. **Preprocessing Step**: Compute a **prefix table (also known as failure function or pi array)** for the pattern. This table will be used to decide the next characters to match.
2. **Matching Step**: Slide the pattern over the text one character at a time. When a mismatch is found, use the prefix table to skip over characters that we know will anyway match.

The preprocessing step takes **O(m)** time, where **m** is the length of the pattern. The matching step takes **O(n)** time, where **n** is the length of the text. Therefore, the KMP algorithm has a time complexity of **O(m + n)**.

The space complexity of the KMP algorithm is **O(m)**, as it requires storing the prefix table for the pattern.

```
function KMP(string s[1..n], string pattern[1..m])
    Compute prefix table for pattern
    i := 0; j := 0
    while i < n
        if s[i] = pattern[j]
            i++; j++
            if j = m
                return i - j (match found)
        else if j > 0
            j := prefix[j-1] (use prefix table to skip characters)
        else
            i++
    return not found
```

8 What is a sliding window? Where this technique is used to solve the programming problems.
ANS 4

9 What are bit manipulation operators? Explain and, or, not, ex-or bit-wise operators.

Bit manipulation operators are used to perform operations at the bit level. These operators are commonly used in low-level programming such as kernel or driver development, as well as in performance-critical code. Here's a brief explanation of the bit manipulation operators you asked about:

1. **AND (&)**: The bitwise AND operator takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.

   For example, 12 (1100 in binary) & 10 (1010 in binary) will give 8 (1000 in binary).
2. **OR (|)**: The bitwise OR operator takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.

   For example, 12 (1100 in binary) | 10 (1010 in binary) will give 14 (1110 in binary).
3. **NOT (~)**: The bitwise NOT operator takes one number and inverts all bits of it.

   For example, ~12 (which is 1100 in binary) will give -13 (which is 0011 in binary in 2's complement form).
4. **XOR (^)**: The bitwise XOR (exclusive OR) operator takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.

   For example, 12 (1100 in binary) ^ 10 (1010 in binary) will give 6 (0110 in binary).

10	Why are the benefits for linked list over arrays. benefits of linked lists over arrays: 1. **Dynamic Size**: Unlike arrays, linked lists do not have a fixed size. This makes them flexible for situations where the amount of data is not known in advance. 2. **Ease of Insertion/Deletion**: Linked lists can insert or delete nodes at any point with O(1) time complexity (if we have a pointer to the node), unlike arrays which require shifting elements. 3. **Efficient Memory Utilization**: Linked lists use memory more efficiently when elements are frequently added and removed. In an array, all elements need to be moved to fill or open up space, which can be costly in terms of time. 4. **No Memory Wastage**: In linked lists, nodes can be allocated as and when required, thus reducing memory waste. In contrast, in arrays, we need to allocate memory beforehand. 5. **Implementation of Stacks and Queues**: Linked lists are used to implement other abstract data types like stacks and queues.

11	Implement singly linked list.
	
	`class Node:`
	`    def __init__(self, data=None):`
	`        self.data = data`
	`        self.next = None`
	
	`class LinkedList:`
	`    def __init__(self):`
	`        self.head = None`
	
	`    def insert(self, data):`
	`        if not self.head:`
	`            self.head = Node(data)`
	`        else:`
	`            cur = self.head`
	`            while cur.next:`
	`                cur = cur.next`
	`            cur.next = Node(data)`
	
	`    def display(self):`
	`        cur = self.head`
	`        while cur:`
	`            print(cur.data, end=' ')`
	`            cur = cur.next`
	`        print()`
	
	`# Usage`
	`ll = LinkedList()`
	`ll.insert(1)`
	`ll.insert(2)`
	`ll.insert(3)`
	`ll.display()  # Outputs: 1 2 3`

11	Implement stack with singly linked list.
	
	`class Node:`
	`    def __init__(self, data=None):`
	`        self.data = data`
	`        self.next = None`
	
	`class Stack:`
	`    def __init__(self):`
	`        self.top = None`
	
	`    def push(self, data):`
	`        new_node = Node(data)`
	`        new_node.next = self.top`

```
        self.top = new_node

    def pop(self):
        if self.top is None:
            return None
        else:
            popped_data = self.top.data
            self.top = self.top.next
            return popped_data

    def peek(self):
        return self.top.data if self.top else None

# Usage
s = Stack()
s.push(1)
s.push(2)
s.push(3)
print(s.pop())  # Outputs: 3
print(s.peek())  # Outputs: 2
```

12 Implement queue with singly linked list.

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None

class Queue:
    def __init__(self):
        self.front = self.rear = None

    def is_empty(self):
        return self.front is None

    def enqueue(self, data):
        new_node = Node(data)
        if self.rear is None:
            self.front = self.rear = new_node
            return
        self.rear.next = new_node
        self.rear = new_node

    def dequeue(self):
        if self.is_empty():
            return None
        temp = self.front
        self.front = temp.next
        if self.front is None:
            self.rear = None
        return temp.data

# Usage
q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
print(q.dequeue())  # Outputs: 1
```

12 Implement doubly linked list.

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None
```

	```
	        self.prev = None

	class DoublyLinkedList:
	    def __init__(self):
	        self.head = None

	    def append(self, data):
	        if self.head is None:
	            self.head = Node(data)
	        else:
	            new_node = Node(data)
	            cur = self.head
	            while cur.next:
	                cur = cur.next
	            cur.next = new_node
	            new_node.prev = cur

	    def prepend(self, data):
	        if self.head is None:
	            self.head = Node(data)
	        else:
	            new_node = Node(data)
	            new_node.next = self.head
	            self.head.prev = new_node
	            self.head = new_node

	    def print_list(self):
	        cur = self.head
	        while cur:
	            print(cur.data)
	            cur = cur.next
	```

13	Implement circular linked list.
	```
	class Node:
	    def __init__(self, data=None):
	        self.data = data
	        self.next = None

	class CircularLinkedList:
	    def __init__(self):
	        self.head = None

	    def append(self, data):
	        if not self.head:
	            self.head = Node(data)
	            self.head.next = self.head
	        else:
	            new_node = Node(data)
	            cur = self.head
	            while cur.next != self.head:
	                cur = cur.next
	            cur.next = new_node
	            new_node.next = self.head

	    def print_list(self):
	        cur = self.head
	        while True:
	            print(cur.data)
	            cur = cur.next
	            if cur == self.head:
	                break
	```

14	Implement circular queue with linked list.

	```
	class Node:
	    def __init__(self, data):
	        self.data = data
	        self.next = None

	class CircularQueue:
	    def __init__(self):
	        self.front = self.rear = None

	    def enqueue(self, data):
	        temp = Node(data)

	        if self.rear == None:
	            self.front = self.rear = temp
	            self.rear.next = self.front
	        else:
	            self.rear.next = temp
	            self.rear = temp
	            self.rear.next = self.front

	    def dequeue(self):
	        if self.front == None:
	            print("Circular Queue is empty")
	            return

	        temp = self.front
	        self.front = temp.next
	        self.rear.next = self.front

	        if self.front == self.rear:
	            self.front = self.rear = None

	    def display(self):
	        temp = self.front
	        while temp:
	            print(temp.data, end=" ")
	            temp = temp.next
	            if temp == self.front:
	                break
	```
15	What is recursion? What is tail recursion? Recursion is a process in which a function calls itself as a subroutine. This allows the function to be broken down into smaller, more manageable problems that are easier to solve1. It’s a method of solving a computational problem where the solution depends on solutions to smaller instances of the same problem2. Recursion has some important properties1: The ability to solve a problem by breaking it down into smaller sub-problems. A recursive function must have a base case or stopping criteria to avoid infinite recursion. Recursive functions may be less efficient than iterative solutions in terms of memory and performance. Tail recursion is a special case of recursion where the recursive call is the last operation in the recursive function3. What this means is that all calculations are performed first, and the execution of the recursive call is the final operation3. This is important because it allows certain compilers to optimize the recursion, using a loop construct instead of a function call for the recursive step3. This can result in significant savings in both processing time and stack space3. Here’s an example of a tail-recursive function in Python:

	Python
	
	`def factorial(n, acc=1):`
	`    if n == 0:`
	`        return acc`
	`    else:`
	`        return factorial(n-1, n*acc)`
	
	In this example, factorial(n-1, n*acc) is the last operation in the recursive function, making it tail recursive. The acc parameter accumulates the result of the factorial calculation, which is returned when n reaches 0. This is the base case of the recursion. The advantage of this tail-recursive version is that it uses constant stack space and is more efficient.

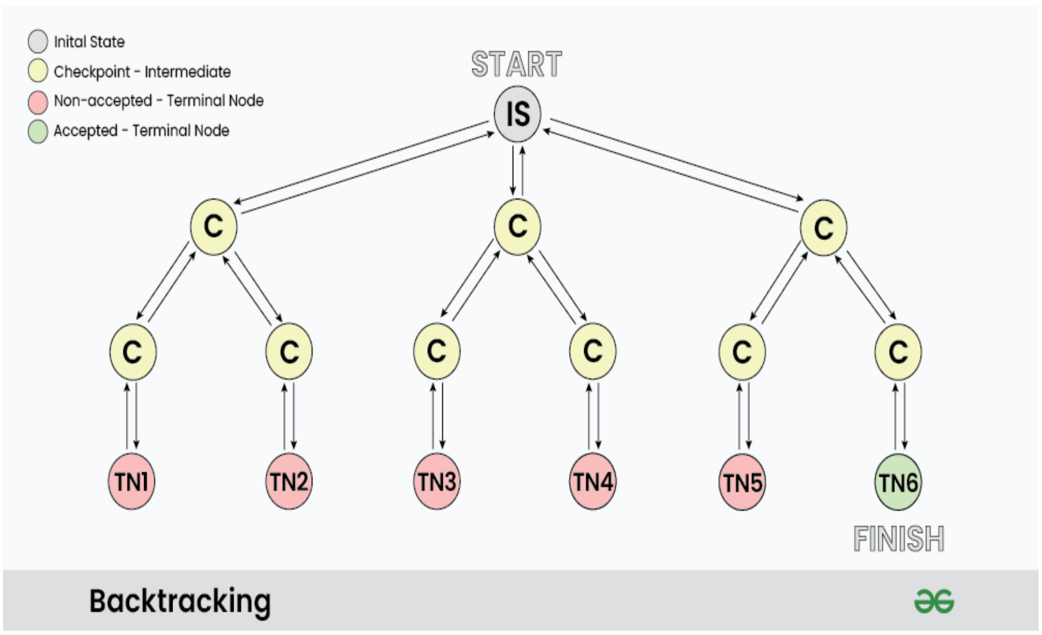
16	What is the tower of Hanoi problem? Write a program to implement the Tower of Hanoi problem. Find the time and space complexity of the program.
	
	The **Tower of Hanoi** problem is a classic problem in the field of computer science and mathematics. It consists of three rods and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.
	
	The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:
	
	1. Only one disk can be moved at a time.
	2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
	3. No disk may be placed on top of a smaller disk.
	
	`def TowerOfHanoi(n , source, destination, auxiliary):`
	`    if n==1:`
	`        print("Move disk 1 from source",source,"to destination",destination)`
	`        return`
	`    TowerOfHanoi(n-1, source, auxiliary, destination)`
	`    print("Move disk",n,"from source",source,"to destination",destination)`
	`    TowerOfHanoi(n-1, auxiliary, destination, source)`
	
	`# Driver code`
	`n = 4`
	`TowerOfHanoi(n,'A','B','C')`
	`# A, C, B are the name of rods`
	
	The **time complexity** of the Tower of Hanoi problem is **O(2^n)**, where n is the number of disks. This is because with each increase in the number of disks, the number of steps required to solve the problem doubles.
	
	The **space complexity** of the Tower of Hanoi problem is **O(n)**, where n is the number of disks. This is due to the recursive nature of the problem, which results in n recursive calls, and thus n items in the call stack. Each recursive call requires a constant amount of space.

17	What is backtracking in algorithms? What kind of problems are solved with this technique? Backtracking is a general algorithmic technique that involves finding a solution incrementally by trying different options and undoing them if they lead to a dead end1. It is commonly used in situations where you need to explore multiple possibilities to solve a problem, like searching for a path in a maze or solving puzzles like Sudoku1. A backtracking algorithm works by recursively exploring all possible solutions to a problem. It

starts by choosing an initial solution, and then it explores all possible extensions of that solution. If an extension leads to a solution, the algorithm returns that solution. If an extension does not lead to a solution, the algorithm backtracks to the previous solution and tries a different extension1.

Backtracking can be applied to a wide range of problems. Here are some examples of problems that can be solved using backtracking:

Puzzles: Such as the N-Queens problem, Sudoku, and the Knight's tour problem12.
Path Finding: Finding the shortest path through a maze or finding all paths from source to destination in a matrix12.
Combinatorial Problems: Generating all possible combinations of a set of items3.
Constraint Satisfaction Problems: Finding a specific solution that satisfies a set of constraints3.
Remember, while backtracking can solve these problems, it may not always be the most efficient method, especially for problems with large input sizes or many potential solutions. Other techniques like dynamic programming might be more suitable in such cases4.



18 Implement N-Queens problem. Find the time and space complexity.

The N-Queens problem is a classic example of a problem that can be solved using backtracking. The problem is to place N queens on an N x N chessboard such that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

```
def isSafe(board, row, col, N):
    # Check this row on left side
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check upper diagonal on left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check lower diagonal on left side
    for i, j in zip(range(row, N, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True

def solveNQUtil(board, col, N):
    # base case: If all queens are placed
    if col >= N:
        return True

    # Consider this column and try placing this queen in all rows one by one
    for i in range(N):
        if isSafe(board, i, col, N):
            # Place this queen in board[i][col]
            board[i][col] = 1

            # recur to place rest of the queens
            if solveNQUtil(board, col + 1, N):
                return True

            # If placing queen in board[i][col] doesn't lead to a solution, then remove queen from
board[i][col]
            board[i][col] = 0

    # If the queen cannot be placed in any row in this column col then return false
    return False

def solveNQ(N):
    board = [[0 for _ in range(N)] for _ in range(N)]

    if not solveNQUtil(board, 0, N):
        print("Solution does not exist")
        return False

    printSolution(board)
    return True

def printSolution(board):
    for i in range(N):
        for j in range(N):
            print(board[i][j], end = " ")
        print()

# Driver Code
N = 4
solveNQ(N)
```

1. `def isSafe(board, row, col, N)`: This function checks if a queen can be placed at board[row][col]. It returns `False` if there's a queen in the same row or in the upper or lower diagonal. Otherwise, it returns `True`.
2. `def solveNQUtil(board, col, N)`: This function is a recursive utility that solves the N-Queens problem. It returns `False` if queens cannot be placed, otherwise, it returns `True` and prints placements in the form of 1s. Please note that there may be more than one solutions, this function prints one of the feasible solutions.
3. `def solveNQ(N)`: This function solves the N-Queens problem using `solveNQUtil()`. It mainly uses `solveNQUtil()` to solve the problem. It returns `False` if queens cannot be placed, otherwise, `True` and prints placements in the form of 1s.
4. `def printSolution(board)`: This function prints the final solution. It prints the board matrix where the queens are placed.
5. `N = 4` This is where you define the size of the board and the number of queens.
6. `solveNQ(N)` This is where the program starts. It calls the `solveNQ` function to solve the problem.

19 What is subset sum problem? Write a recursive function to solve the subset sum problem?

The **Subset Sum Problem** is a classic computer science problem that falls under the category of **NP-Complete problems**. It is a decision problem that can be stated as follows:

Given a set of integers and an integer **S**, does any subset of the given set add up to **S**?

```
def is_subset_sum(set, n, sum):
    # Base Cases
    if sum == 0:
        return True
    if n == 0 and sum != 0:
        return False

    # If last element is greater than sum, then ignore it
    if set[n-1] > sum:
        return is_subset_sum(set, n-1, sum)

    # Else, check if sum can be obtained by any of the following:
    # (a) including the last element
    # (b) excluding the last element
    return is_subset_sum(set, n-1, sum) or is_subset_sum(set, n-1, sum-set[n-1])

# Test the function
set = [3, 34, 4, 12, 5, 2]
sum = 9
n = len(set)
if (is_subset_sum(set, n, sum) == True):
    print("Found a subset with given sum")
else:
    print("No subset with given sum")
```

20 Implement a function that uses the sliding window technique to find the maximum sum of any contiguous subarray of size K.

```
def max_sum_subarray_size_k(arr, k):
    window_sum = 0
    max_sum = 0
    window_start = 0

    for window_end in range(len(arr)):
        window_sum += arr[window_end]  # add the next element

        # slide the window, we don't need to slide if we've not hit the required window size of 'k'
        if window_end >= k-1:
            max_sum = max(max_sum, window_sum)
            window_sum -= arr[window_start]  # subtract the element going out
            window_start += 1  # slide the window ahead

    return max_sum

# Test the function
arr = [2, 3, 4, 1, 5]
k = 3
print("Maximum sum of a subarray of size K: ", max_sum_subarray_size_k(arr, k))
```

21 Write a recursive function to generate all possible subsets of a given set.

```
def generate_subsets(s, current_subset=[], index=0):
    if index == len(s):
        print(current_subset)
    else:
        # Two possibilities for each element, it's either in the subset or not
        # First, we select the element to be in the subset
        generate_subsets(s, current_subset + [s[index]], index + 1)
        # Then, we do not select the element to be in the subset
        generate_subsets(s, current_subset, index + 1)

# Test the function
s = [1, 2, 3]
generate_subsets(s)
```

22 Write a program to find the first occurrence of repeating character in a given string.

```
def first_repeating_char(s):
    char_count = {}
    for char in s:
        if char in char_count:
            return char
        else:
            char_count[char] = 1
    return None

# Test the function
s = "interviewquery"
print("The first repeating character is: ", first_repeating_char(s))
```

23 Write a program to print all the LEADERS in the array. An element is a leader if it is greater than all the elements to its right side. And the rightmost element is always a leader.

```
def print_leaders(arr):
    max_from_right = arr[-1]  # The rightmost element is always a leader
    print(max_from_right, end=' ')
    for i in range(len(arr)-2, -1, -1):
        if max_from_right <= arr[i]:  # If this element is greater than the max from
right
            print(arr[i], end=' ')  # This element is a leader
            max_from_right = arr[i]  # Update the max from right

# Test the function
arr = [16, 17, 4, 3, 5, 2]
print("Leaders in the array are: ", end='')
print_leaders(arr)
```

24 Write a program to find the majority element in the array. A majority element in an array A[] of size n is an element that appears more than n/2 times.

```
def find_majority_element(arr):
    majority_element = 0
    count = 0

    # Boyer-Moore Voting Algorithm
    for i in range(len(arr)):
        if count == 0:
            majority_element = arr[i]
        if arr[i] == majority_element:
            count += 1
        else:
            count -= 1

    # Verify if the majority_element appears more than n/2 times
    count = arr.count(majority_element)
    if count > len(arr) // 2:
        return majority_element
    else:
        return "No Majority Element"

# Test the function
arr = [2, 2, 1, 1, 1, 2, 2]
print(find_majority_element(arr))
```

25 Given an integer k and a queue of integers, write a program to reverse the order of the first k elements of the queue, leaving the other elements in the same relative order.

```
from queue import Queue
from collections import deque

def reverse_first_k(queue, k):
    if queue.empty() or k > queue.qsize():
        return "Invalid input"

    stack = deque()
    for _ in range(k):
        stack.append(queue.get())
    while stack:
        queue.put(stack.pop())
    for _ in range(queue.qsize() - k):
        queue.put(queue.get())

    return queue

# Test the function
queue = Queue()
for i in range(1, 11):
    queue.put(i)

reverse_first_k(queue, 5)

while not queue.empty():
    print(queue.get(), end=' ')
```

26 Write a program to implement a stack using queues.

```
class Stack:
    def __init__(self):
        self.q1 = []
        self.q2 = []

    def push(self, x):
        self.q1.append(x)

    def pop(self):
        if not self.q1:
            return "Stack is empty"
        while len(self.q1) > 1:
            self.q2.append(self.q1.pop(0))
        item = self.q1.pop()
        self.q1, self.q2 = self.q2, self.q1
        return item
```

```
# Test the Stack
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)
print(stack.pop())  # prints 3
print(stack.pop())  # prints 2
```

27 Wrire a program to implement queue using stacks.

```
class Queue:
    def __init__(self):
        self.s1 = []
        self.s2 = []

    def enqueue(self, x):
        self.s1.append(x)

    def dequeue(self):
        if not self.s1 and not self.s2:
            return "Queue is empty"
        if not self.s2:
            while self.s1:
                self.s2.append(self.s1.pop())
        return self.s2.pop()

# Test the Queue
queue = Queue()
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
print(queue.dequeue())  # prints 1
print(queue.dequeue())  # prints 2
```

28 Given a string S of lowercase alphabets, write a program to check if string is isogram or not. An Isogram is a string in which no letter occurs more than once.

```
def is_isogram(string):
    # Convert the string to lowercase for uniformity
    string = string.lower()

    # Create a set from the string. In a set, duplicate elements are not allowed.
    # So, if the length of the set is equal to the length of the string, it means all characters were unique.
    return len(string) == len(set(string))

# Test the function
print(is_isogram("subdermatoglyphic"))  # This should return True
print(is_isogram("hello"))  # This should return False
```

29 Given a sorted array, arr[] consisting of N integers, write a program to find the frequencies of each array element.

```
def count_frequencies(arr):
    # Initialize the count and the element
    count = 1
    element = arr[0]

    # Iterate over the array
    for i in range(1, len(arr)):
        # If the current element is the same as the previous one, increment the
count
        if arr[i] == element:
            count += 1
        else:
            # Print the element and its frequency
            print(f"Element {element} occurs {count} times")
```

	```
	            # Update the element and reset the count
	            element = arr[i]
	            count = 1

	    # Print the last element and its frequency
	    print(f"Element {element} occurs {count} times")

	# Test the function
	arr = [1, 1, 2, 2, 2, 3, 4, 4, 4, 4]
	count_frequencies(arr)
	```

30	Write a program to delete middle element from stack.
	```
	def delete_middle(stack, n, curr=0):
	    # Base case: If stack is empty or all items are traversed
	    if not stack or curr == n:
	        return

	    # Remove current item
	    temp = stack.pop()

	    # Remove remaining items
	    delete_middle(stack, n, curr+1)

	    # Put all items back except middle
	    if curr != n//2:
	        stack.append(temp)

	# Test the function
	stack = [1, 2, 3, 4, 5]
	delete_middle(stack, len(stack))
	print(stack)  # This should print [1, 2, 4, 5]
	```

31	Write a program to remove consecutive duplicates from string.
	```
	def remove_consecutive_duplicates(s):
	    # Initialize the result
	    result = ""

	    # Iterate over the string
	    for i in range(len(s)):
	        # If the current character is not the same as the previous one, add it to the result
	        if i == 0 or s[i] != s[i-1]:
	            result += s[i]

	    return result

	# Test the function
	print(remove_consecutive_duplicates("aaabbbccdaaa"))  # This should print "abcda"
	```

33	Write a program to display next greater element of all element given in array.
	```
	def print_next_greater(arr):
	    stack = []
	    for i in range(len(arr)):
	        while stack and stack[-1] < arr[i]:
	            print(f"Next greater element for {stack.pop()} is {arr[i]}")
	        stack.append(arr[i])
	    while stack:
	        print(f"Next greater element for {stack.pop()} is -1")

	# Test the function
	arr = [4, 5, 2, 25]
	print_next_greater(arr)
	```

34	Write a program to evaluate a postfix expression.

```
def evaluate_postfix(expression):
    stack = []
    for char in expression:
        if char.isdigit():
            stack.append(int(char))
        else:
            operand2 = stack.pop()
            operand1 = stack.pop()
            if char == '+':
                result = operand1 + operand2
            elif char == '-':
                result = operand1 - operand2
            elif char == '*':
                result = operand1 * operand2
            elif char == '/':
                result = operand1 / operand2
            stack.append(result)
    return stack[0]

# Test the function
expression = "231*+9-"
print(evaluate_postfix(expression))
```

35

Write a program to get MIN at pop from stack.

```
class MinStack:
    def __init__(self):
        self.stack = []
        self.min_stack = []

    def push(self, x):
        self.stack.append(x)
        if not self.min_stack or x <= self.min_stack[-1]:
            self.min_stack.append(x)

    def pop(self):
        if self.stack:
            top = self.stack.pop()
            if self.min_stack and self.min_stack[-1] == top:
                self.min_stack.pop()
            return top

    def get_min(self):
        return self.min_stack[-1] if self.min_stack else None

# Test the MinStack
min_stack = MinStack()
min_stack.push(3)
min_stack.push(5)
print(min_stack.get_min())  # prints 3
print(min_stack.pop())  # prints 5
print(min_stack.get_min())  # prints 3
```

36

Write a program to swap kth node from ends in given single linked list.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

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

    def print_list(self):
        node = self.head
        while node:
            print(node.data, end=" ")
            node = node.next
        print()

    def swap_kth_node(self, k):
        n = 0
        node = self.head
        while node:
            n += 1
            node = node.next

        if n < k:
            return

        x = self.head
        x_prev = None
        for i in range(k - 1):
            x_prev = x
            x = x.next

        y = self.head
        y_prev = None
        for i in range(n - k):
            y_prev = y
            y = y.next

        if x_prev:
            x_prev.next = y
        else:
            self.head = y

        if y_prev:
            y_prev.next = x
        else:
            self.head = x

        temp = x.next
        x.next = y.next
        y.next = temp

# Test the LinkedList
llist = LinkedList()
for i in range(8, 0, -1):
    llist.push(i)
print("Original linked list:")
llist.print_list()
llist.swap_kth_node(3)
print("Linked list after swapping 3rd node from both ends:")
llist.print_list()
```

37 Write a program to detect loop in linked list

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

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

    def detect_loop(self):
        slow_p = self.head
        fast_p = self.head
        while slow_p and fast_p and fast_p.next:
            slow_p = slow_p.next
            fast_p = fast_p.next.next
            if slow_p == fast_p:
                return True
        return False

# Test the LinkedList
llist = LinkedList()
for i in range(5, 0, -1):
```

	`    llist.push(i)`
	`# Create a loop for testing`
	`llist.head.next.next.next.next.next = llist.head.next.next`
	`print("Loop detected:" if llist.detect_loop() else "No loop detected")`

38	Write a program to find Intersection point in Y shaped Linked list.
	
	`class Node:`
	`    def __init__(self, data):`
	`        self.data = data`
	`        self.next = None`
	
	`def getIntersectionNode(head1, head2):`
	`    curr1 = head1`
	`    curr2 = head2`
	
	`    # Traverse through both lists. If one list ends, start at the beginning of the other list.`
	`    # If there is an intersection, the pointers should meet at the intersection point after 2`
	`traversals.`
	`    while curr1 != curr2:`
	`        curr1 = curr1.next if curr1 else head2`
	`        curr2 = curr2.next if curr2 else head1`
	
	`    return curr1.data  # Return the intersection node`
	
	`# Test the function with an example`
	`# Create a Y shaped linked list`
	`head1 = Node(3)`
	`head1.next = Node(6)`
	`head1.next.next = Node(9)`
	`head1.next.next.next = Node(15)`
	`head1.next.next.next.next = Node(30)`
	
	`head2 = Node(10)`
	`head2.next = head1.next.next.next  # Intersection point`
	
	`print("Intersection point is", getIntersectionNode(head1, head2))`

39	Write a program to merge two sorted linked list.
	
	`class Node:`
	`    def __init__(self, data):`
	`        self.data = data`
	`        self.next = None`
	
	`def mergeLists(head1, head2):`
	`    # A dummy node to store the result`
	`    dummyNode = Node(0)`
	
	`    # last stores the last node`
	`    last = dummyNode`
	`    while True:`
	
	`        # If either list runs out, use the other list`
	`        if head1 is None:`
	`            last.next = head2`
	`            break`
	`        if head2 is None:`
	`            last.next = head1`
	`            break`
	
	`        # Compare the data of the lists and whichever is smaller is appended to the last's`
	`next and the head is changed`

```
        if head1.data <= head2.data:
            last.next = head1
            head1 = head1.next
        else:
            last.next = head2
            head2 = head2.next

        # Update the last for next insertion
        last = last.next

    # Returns the head of the merged list
    return dummyNode.next

# Test the function with an example
# Create two sorted linked lists
head1 = Node(1)
head1.next = Node(2)
head1.next.next = Node(4)

head2 = Node(1)
head2.next = Node(3)
head2.next.next = Node(4)

merged_head = mergeLists(head1, head2)

# Print the merged list
while merged_head is not None:
    print(merged_head.data, end=" ")
    merged_head = merged_head.next
```

40 Write a program to find max and second max of array.

```
def find_max_and_second_max(arr):
    if len(arr) < 2:
        return "Invalid input. Array should have at least two elements."

    max1 = max2 = float('-inf')
    for num in arr:
        if num > max1:
            max2 = max1
            max1 = num
        elif num > max2 and num != max1:
            max2 = num

    if max2 == float('-inf'):
        return "No second max value, all elements are the same."
    else:
        return max1, max2

# Test the function with an example
arr = [2, 3, 6, 6, 5]
print("Max and second max of array:", find_max_and_second_max(arr))
```

41 Write a program to find Smallest Positive missing number. You are given an array arr[] of N integers. The task is to find the smallest positive number missing from the array. Positive number starts from 1.

```
def find_smallest_missing_positive(arr):
    if not arr:
        return 1

    arr = set(arr)
    smallest_positive = 1

    while smallest_positive in arr:
```

	```
	        smallest_positive += 1

	    return smallest_positive

	# Test the function with an example
	arr = [0, 10, 2, -10, -20, 1, 3]
	print("Smallest positive missing number:", find_smallest_missing_positive(arr))
	```

42	Given a non-negative integer N. The task is to check if N is a power of 2. More formally, check if N can be expressed as 2x for some integer x. Return true if N is power of 2 else return false.
	```
	def is_power_of_two(n):
	    if n <= 0:
	        return False
	    else:
	        return (n & (n - 1)) == 0

	# Test the function with an example
	n = 16
	print("Is", n, "a power of 2?", is_power_of_two(n))
	```
	1. **If `n` is less than or equal to 0**: The function returns `False`. This is because powers of 2 are always positive and non-zero.
	2. **If `n` is greater than 0**: The function checks if `n` AND `n-1` is equal to 0. This is a bitwise operation.
	  ○ The bitwise AND operation (`&`) compares each bit of the first operand (`n`) to the corresponding bit of the second operand (`n-1`). If both bits are 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.
	  ○ For numbers that are powers of 2, their binary representation has a single '1' bit and the rest are '0'. For example, 2 is `10` in binary and 4 is `100` in binary.
	  ○ When you subtract 1 from these numbers, you get a binary number with the '1' bit turned to '0' and all bits to the right of it turned to '1'. For example, `2-1` is `1` which is `01` in binary and `4-1` is `3` which is `011` in binary.
	  ○ So, when you perform a bitwise AND operation between a power of 2 and one less than it, you get `0` because there are no positions where both numbers have a '1' bit.
	  ○ Therefore, `(n & (n - 1)) == 0` is a quick way to check if a number is a power of 2.

43	Write a program to Count Total Digits in a Number using recursion. You are given a number n. You need to find the count of digits in n.
	```
	def count_digits(n):
	    if n == 0:
	        return 0
	    return 1 + count_digits(n // 10)

	# Test the function with an example
	n = 12345
	print("The number of digits in", n, "is", count_digits(n))
	```

44	Check whether K-th bit is set or not. Given a number **N** and a bit number **K**, check if **Kth** index bit of **N** is set or not. A bit is called set if it is 1. Position of set bit '1' should be indexed starting with 0 from LSB side in binary representation of the number. Index is starting from 0. You just need to return **true** or **false**.

	```
	def is_kth_bit_set(n, k):
	    if n & (1 << k):
	        return True
	    else:
	        return False

	# Test the function with an example
	n = 5  # binary: 101
	k = 2
	print("Is the", k, "th bit of", n, "set?", is_kth_bit_set(n, k))
	```

45	Write a program to print 1 To N without loop
	```
	def print_numbers(n):
	    if n > 0:
	        print_numbers(n - 1)
	        print(n)

	# Test the function with an example
	N = 10
	print("Printing numbers from 1 to", N)
	print_numbers(N)
	```