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

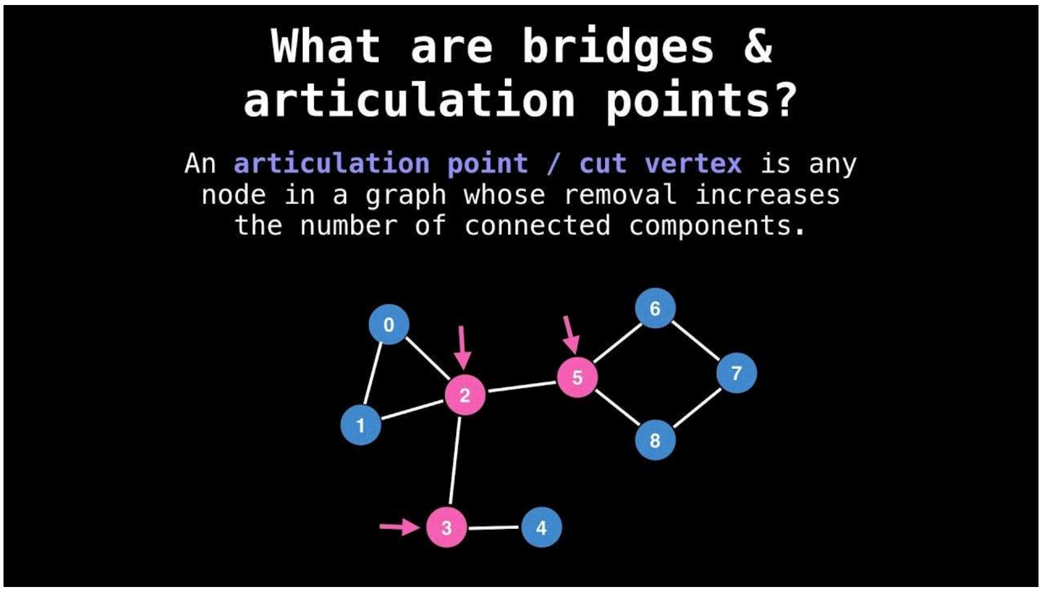
1. **Kruskal's Algorithm**: Builds MST by adding smallest edges without cycles. Applications: Network design, clustering.
2. **Prim's Algorithm**: Grows MST from an arbitrary vertex. Applications: Communication networks, road network optimization.

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.

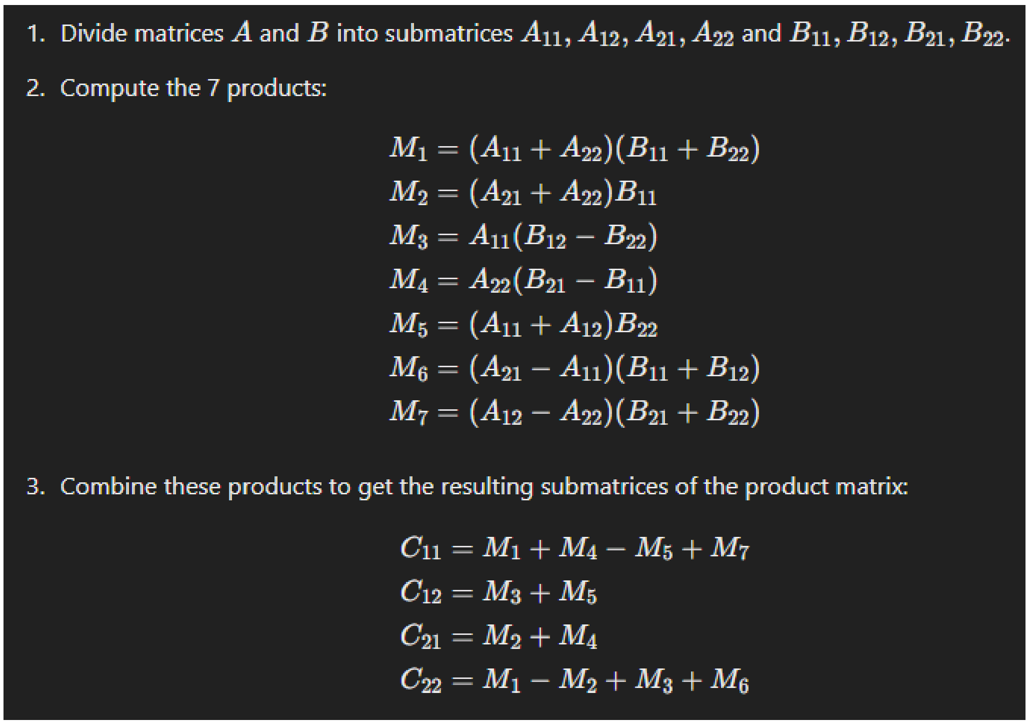
Conventional Matrix Multiplication

- **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

Advanced Algorithmic Problem Solving
mid term answers

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)
```