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 concepts used in algorithm analysis to understand the performance characteristics of an algorithm.

Time Complexity:

Time complexity represents the amount of time an algorithm takes to run as a function of the length of the input. It measures the number of basic operations (such as comparisons, assignments, arithmetic operations, etc.) performed by the algorithm relative to the size of the input. Time complexity is typically expressed using Big O notation, which provides an upper bound on the growth rate of the algorithm's running time as the input size increases.

For example:

- An algorithm with time complexity O(1) (constant time) means that its running time does not depend on the size of the input.
- An algorithm with time complexity O(n) (linear time) means that its running time increases linearly with the size of the input.
- An algorithm with time complexity O(n^2) (quadratic time) means that its running time increases quadratically with the size of the input.

Space Complexity:

Space complexity represents the amount of memory space required by an algorithm to solve a problem as a function of the size of the input. It measures the maximum amount of memory used by the algorithm, including variables, data structures, and other resources, relative to the size of the input. Like time complexity, space complexity is also typically expressed using Big O notation.

For example:

- An algorithm with space complexity O(1) (constant space) means that its memory usage remains constant regardless of the size of the input.
- An algorithm with space complexity O(n) (linear space) means that its memory usage increases linearly with the size of the input.
- An algorithm with space complexity O(n^2) (quadratic space) means that its memory usage increases quadratically with the size of the input.

In summary, time complexity and space complexity are used to analyze and compare the efficiency and scalability of algorithms in terms of their running time and memory usage, respectively.

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

Asymptotic notations are mathematical notations used to describe the behavior of functions as their input values approach infinity. They are commonly used in algorithm analysis to express the growth rates of algorithms' time and space complexities in terms of the input size.

Here are the commonly used asymptotic notations:

1. Big O (O):
   - Big O notation represents the upper bound of an algorithm's time or space complexity. It provides an upper limit on the growth rate of the function.
   - Formally, a function f(n) is said to be O(g(n)) if there exist positive constants c and n0 such that $f(n) \leq c \; g(n)$ for all $n \geq n0$.
   - In simple terms, O(g(n)) represents the maximum rate of growth of the function f(n), up to a constant factor, as n becomes large.

2. Omega (Ω):
   - Omega notation represents the lower bound of an algorithm's time or space complexity. It provides a lower limit on the growth rate of the function.
   - Formally, a function f(n) is said to be Ω(g(n)) if there exist positive constants c and n0 such that $f(n) \geq c \; g(n)$ for all $n \geq n0$.
   - In simple terms, Ω(g(n)) represents the minimum rate of growth of the function f(n), up to a constant factor, as n becomes large.

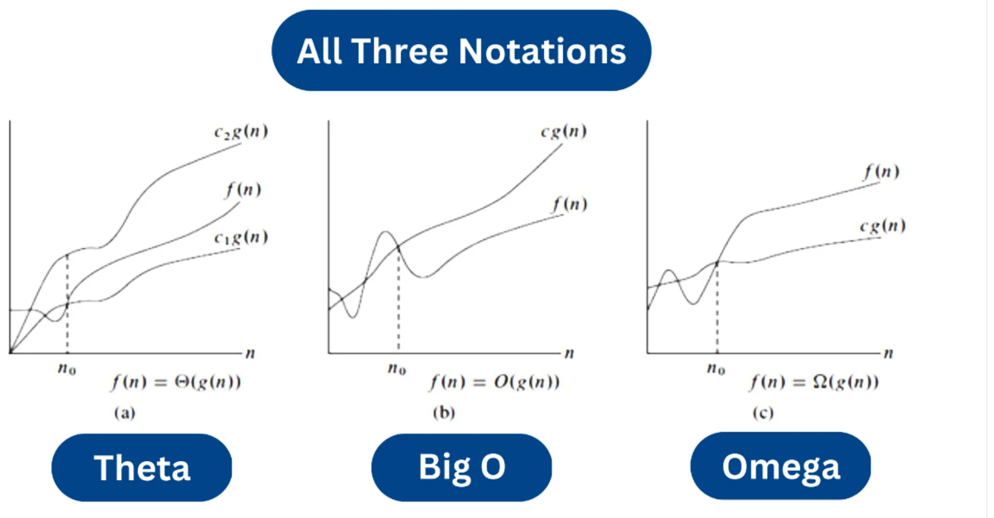
3. Theta (Θ):
   - Theta notation represents both the upper and lower bounds of an algorithm's time or space complexity. It provides a tight bound on the growth rate of the function.
   - Formally, a function f(n) is said to be Θ(g(n)) if there exist positive constants c1, c2, and n0 such that $c1 \; g(n) \leq f(n) \leq c2 \; g(n)$ for all $n \geq n0$.
   - In simple terms, Θ(g(n)) represents the rate of growth of the function f(n) that matches

the rate of growth of g(n) within constant factors, as n becomes large.

4. Small omega (ω):
 - Small omega notation represents a stronger lower bound than Omega notation. It provides an indication that a function grows faster than another function.
 - Formally, a function f(n) is said to be ω(g(n)) if for any positive constant c, there exists an integer n0 such that f(n) ≥ c g(n) for all n ≥ n0.
 - In simple terms, ω(g(n)) represents that the function f(n) grows strictly faster than the function g(n) as n becomes large.

5. Small o (o):
 - Small o notation represents a stronger upper bound than Big O notation. It provides an indication that a function grows slower than another function.
 - Formally, a function f(n) is said to be o(g(n)) if for any positive constant c, there exists an integer n0 such that f(n) ≤ c g(n) for all n ≥ n0.
 - In simple terms, o(g(n)) represents that the function f(n) grows strictly slower than the function g(n) as n becomes large.

In summary, these asymptotic notations help in describing the behavior and efficiency of algorithms by providing insights into their growth rates relative to the input size.



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

Sure, let's break down the meaning of each of these time complexities:

1. O(2^n):

- This notation represents an exponential time complexity, where the running time of the algorithm grows exponentially with the size of the input.
- It indicates that the algorithm's running time doubles with each additional input element.
- Example: A classic example of an algorithm with O(2^n) time complexity is the recursive solution to the Fibonacci sequence. Each recursive call results in two more recursive calls until it reaches the base cases.

2. O(n^2):
   - This notation represents a quadratic time complexity, where the running time of the algorithm grows quadratically with the size of the input.
   - It indicates that the algorithm's running time increases quadratically with the number of input elements.
   - Example: An example of an algorithm with O(n^2) time complexity is the bubble sort algorithm. In the worst-case scenario, where the array is sorted in reverse order, bubble sort performs n(n-1)/2 comparisons and swaps.

3. O(n log n):
   - This notation represents a logarithmic-linear time complexity, commonly seen in efficient sorting algorithms like merge sort and heap sort.
   - It indicates that the algorithm's running time grows in proportion to n multiplied by the logarithm of n.
   - Example: Merge sort has O(n log n) time complexity. It divides the array into halves recursively until each sub-array contains a single element, then merges the sorted sub-arrays. The divide-and-conquer strategy leads to a time complexity of O(n log n).

4. O(log n):
   - This notation represents a logarithmic time complexity, where the running time of the algorithm grows logarithmically with the size of the input.
   - It indicates that the algorithm's running time increases logarithmically as the input size increases.
   - Example: An example of an algorithm with O(log n) time complexity is binary search. In each step, binary search reduces the search space by half until it finds the target element or determines it doesn't exist.

In summary, each of these time complexities describes how the running time of an algorithm scales with the size of the input. Exponential and quadratic time complexities indicate inefficient algorithms for large inputs, while logarithmic-linear and logarithmic time complexities suggest more efficient algorithms.

4. **Explain sliding window protocol.**

The sliding window protocol is a method used in computer networking and communication systems to improve the efficiency of data transmission over unreliable or congested channels. It allows multiple packets to be transmitted without waiting for acknowledgment for each individual packet. This helps in utilizing the available bandwidth more effectively and reducing the overall latency.

Here's how the sliding window protocol works:

1. Sender and Receiver:
   - The communication involves a sender and a receiver. The sender is responsible for sending data packets, and the receiver is responsible for receiving and acknowledging them.

2. Window:
   - The sliding window protocol operates with a "window" concept. The window represents a range of sequence numbers for packets that can be sent or acknowledged at any given time.

3. Sender's Perspective:
   - The sender maintains a "send window" that indicates which packets it can send to the receiver.
   - Initially, the sender's window starts at sequence number 0. It can send packets up to a certain window size.
   - As packets are sent and acknowledged by the receiver, the sender's window slides forward, allowing it to send new packets.

4. Receiver's Perspective:
   - The receiver maintains a "receive window" that indicates which packets it is ready to receive.
   - Initially, the receiver's window starts at sequence number 0. It can receive packets up

to a certain window size.
- As packets are received in order, the receiver's window slides forward, indicating that it is ready to receive the next packet.

5. Acknowledgment:
- When the receiver receives a packet, it sends an acknowledgment (ACK) back to the sender, indicating the highest sequence number it has received successfully.
- The sender uses this acknowledgment to determine which packets have been successfully received by the receiver.
- If the sender does not receive an acknowledgment for a packet within a certain timeout period, it retransmits the packet.

6. Sliding Window Operation:
- As packets are successfully transmitted and acknowledged, both the sender's and receiver's windows slide forward, allowing new packets to be sent and received.
- The size of the window can dynamically adjust based on network conditions, bandwidth availability, and other factors.

The sliding window protocol helps in achieving efficient and reliable data transmission by allowing multiple packets to be in transit simultaneously. It reduces the overhead associated with waiting for acknowledgments for each individual packet, leading to better utilization of network resources and improved throughput.

----

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

The Naive String-Matching algorithm, also known as the Brute-Force algorithm, is a simple and straightforward method for finding all occurrences of a pattern (substring) within a given text (string). It works by checking all possible alignments of the pattern within the text and comparing characters one by one.

Here's how the Naive String-Matching algorithm works:

1. Start with the first character of the text.
2. Compare this character with the first character of the pattern.
3. If they match, continue comparing subsequent characters of the text and pattern.
4. If any characters don't match, move to the next character in the text and repeat steps 2 and 3.

5. Repeat this process until either the entire pattern matches the substring in the text or the end of the text is reached.
6. If the pattern is found in the text, record its starting position/index.

The Naive String-Matching algorithm has a time complexity of O((n - m + 1) m), where:
- n is the length of the text.
- m is the length of the pattern.

The worst-case time complexity occurs when the pattern occurs at every position of the text, resulting in O((n - m + 1) m) comparisons.

The space complexity of the Naive String-Matching algorithm is O(1) because it does not require any additional data structures beyond the input text and pattern. It performs comparisons directly on the characters of the text and pattern without requiring additional memory allocation.

Despite its simplicity, the Naive String-Matching algorithm is not always the most efficient choice for large texts or patterns due to its quadratic time complexity. However, it serves as a baseline algorithm and can be useful for small inputs or as a reference for more sophisticated string-matching algorithms.

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

The Rabin-Karp String-Matching algorithm is a string-searching algorithm that efficiently finds all occurrences of a pattern (substring) within a given text (string). It works by employing hashing to compare the hash values of the pattern and substrings of the text, allowing for faster comparisons.

Here's how the Rabin-Karp algorithm works:

1. Hashing: First, the hash value of the pattern and the hash values of all possible substrings of the text with the same length as the pattern are computed. This is typically done using a rolling hash function, such as the Rabin fingerprint or polynomial rolling hash function.

2. Comparison: Compare the hash value of the pattern with the hash values of substrings of the text. If the hash values match, perform a character-by-character comparison to

confirm the match.

3. Rolling Hash: If the hash values don't match, move to the next substring of the text by "rolling" the hash value. This involves updating the hash value using a rolling hash function as the window slides to the right.

4. Collision Handling: If there's a hash collision (i.e., two different substrings have the same hash value), perform a character-by-character comparison to confirm the match.

5. Repeat: Continue the process until all substrings of the text have been compared with the pattern.

The Rabin-Karp algorithm has a time complexity of O((n - m + 1) m), where:
- n is the length of the text.
- m is the length of the pattern.

The worst-case time complexity occurs when the hash values of all possible substrings of the text need to be computed and compared with the hash value of the pattern.

The space complexity of the Rabin-Karp algorithm is O(1), as it does not require any additional data structures beyond the input text and pattern. However, the hash table or array used for computing and storing the hash values of substrings may require additional space, typically O(n) or O(m), depending on the implementation.

Overall, the Rabin-Karp algorithm can be an efficient string-matching algorithm, especially when dealing with multiple pattern searches in the same text or when the pattern length is relatively small compared to the text length. However, it may not be as efficient as other algorithms for single pattern searches in large texts due to its quadratic worst-case time complexity.
The running time of **RABIN-KARP-MATCHER** in the worst case scenario **O ((n-m+1) m** but it has a good average case running time. If the expected number of strong shifts is small **O (1)** and prime q is chosen to be quite large, then the Rabin-Karp algorithm can be expected to run in time **O (n+m)** plus the time to require to process spurious hits.

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

Advanced Algorithmic Problem Solving (R1UC601B)

The Knuth-Morris-Pratt (KMP) algorithm is a string-searching algorithm that efficiently finds all occurrences of a pattern (substring) within a given text (string). It works by utilizing information about the pattern itself to avoid unnecessary comparisons during the search process.

Here's how the Knuth-Morris-Pratt algorithm works:

1. Preprocessing (Building the Partial Match Table):
   - Construct a partial match table (also known as the "failure function" or "prefix function") for the pattern. This table helps in determining how many characters to skip when a mismatch occurs during the search process.
   - The partial match table is built based on the pattern itself and contains information about the longest proper prefix that is also a suffix for each prefix of the pattern.

2. Searching:
   - Start comparing characters of the text with characters of the pattern from left to right.
   - When a mismatch occurs, use the information from the partial match table to determine how many characters can be skipped in the text without missing a potential match.
   - Update the comparison position in the pattern accordingly and continue the search until either a match is found or the end of the text is reached.

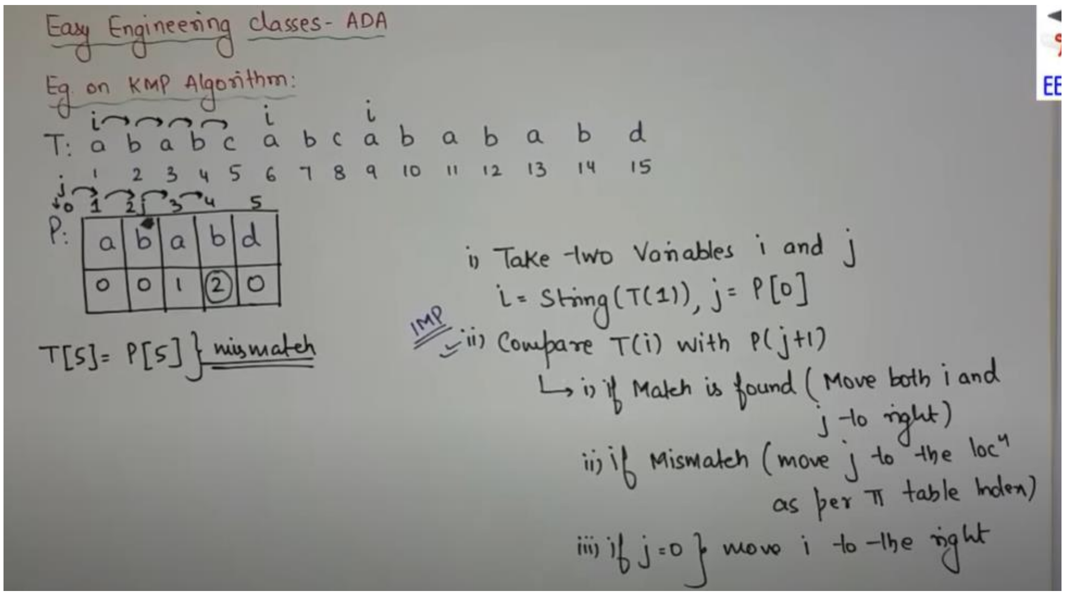
3. Handling Matches:
   - If a complete match is found, record the starting position/index of the match in the text.
   - Continue the search process to find all occurrences of the pattern in the text.

The Knuth-Morris-Pratt algorithm has a time complexity of O(n + m), where:
- n is the length of the text.
- m is the length of the pattern.

The preprocessing step takes O(m) time to construct the partial match table, and the searching step takes O(n) time to compare characters of the text with the pattern while utilizing the information from the partial match table to avoid unnecessary comparisons.

The space complexity of the Knuth-Morris-Pratt algorithm is O(m), where m is the length of the pattern. This space is required to store the partial match table, which contains information about the longest proper prefix that is also a suffix for each prefix of the pattern.

Overall, the Knuth-Morris-Pratt algorithm is an efficient string-matching algorithm, especially for cases where the pattern length is relatively small compared to the text length. It avoids unnecessary character comparisons during the search process by utilizing information from the partial match table, leading to improved performance compared to brute-force methods.


Easy Engineering classes- ADA

Eg. on KMP Algorithm:

T:	a	b	a	b	c	a	b	c	a	b	a	b	a	b	d
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

j	1	2	3	4	5
P:	a	b	a	b	d
	0	0	1	2	0

T[5] = P[5] } mismatch

i) Take two Variables i and j
i = String(T(1)), j = P[0]

IMP ii) Compare T(i) with P(j+1)
↳ i) if Match is found (Move both i and j to right)
ii) if Mismatch (move j to the locⁿ as per π table index)
iii) if j=0 } move i to the right

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

A sliding window is a technique used in programming to efficiently solve problems that involve finding a subarray (or substring) of fixed length or satisfying certain conditions within a larger array (or string). It involves iterating through the array or string with a window of fixed size, adjusting the window as needed to solve the problem.

Here's how the sliding window technique works:

1. Initialize Window: Start with a window of fixed size, typically defined by two pointers (left and right) initially positioned at the beginning of the array or string.

2. Expand and Contract Window: Move the right pointer to expand the window, adding elements to it. If the window meets certain conditions, keep expanding it. Otherwise, move the left pointer to contract the window, removing elements from it.

3. Update Solution: While expanding and contracting the window, keep track of the solution to the problem. This could involve computing the maximum, minimum, sum, or other properties of the elements within the window.

4. Repeat: Continue iterating through the array or string until the right pointer reaches the end.

The sliding window technique is used to efficiently solve a variety of programming problems, including but not limited to:

1. Subarray/Substring Problems:
   - Finding the maximum or minimum sum subarray of fixed length.
   - Finding the longest subarray with at most k distinct elements.
   - Finding the longest substring with at most k distinct characters.

2. String Matching Problems:
   - Finding all occurrences of a pattern (substring) within a larger string.
   - Finding the shortest substring containing all characters of a given pattern.

3. Two Pointer Problems:
   - Finding pairs of elements with a given sum in a sorted or unsorted array.
   - Finding the triplet with a given sum in a sorted or unsorted array.

4. Window Sliding Problems:
   - Finding the maximum sum of a fixed-length subarray.
   - Finding the longest continuous subarray with sum at most k.
   - Finding the smallest subarray with sum greater than or equal to a given value.

Overall, the sliding window technique is a powerful tool for solving a wide range of programming problems efficiently by avoiding redundant computations and reducing the time complexity of the solution. It is especially useful when dealing with problems involving arrays, strings, or sequences of data.

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

Bit manipulation operators are used to perform bitwise operations on individual bits of binary numbers. These operators manipulate the binary representation of numbers at the bit level, allowing for efficient handling of data at the lowest level of abstraction.

Here are the commonly used bitwise operators:

1. Bitwise AND (&):
   - The bitwise AND operator compares corresponding bits of two operands. If both bits are 1, the result is 1; otherwise, it's 0.
   - Symbol: &
   - Example:
     - 0b1010 & 0b1100 = 0b1000

2. Bitwise OR (|):
   - The bitwise OR operator compares corresponding bits of two operands. If either bit is 1, the result is 1; otherwise, it's 0.
   - Symbol: |
   - Example:
     - 0b1010 | 0b1100 = 0b1110

3. Bitwise NOT (~):
   - The bitwise NOT operator (also known as bitwise complement or one's complement) flips all bits of the operand. It converts 0 to 1 and 1 to 0.
   - Symbol: ~
   - Example:
     - ~0b1010 = 0b0101

4. Bitwise XOR (exclusive OR) (^):
   - The bitwise XOR operator compares corresponding bits of two operands. If the bits are different, the result is 1; if they are the same, the result is 0.
   - Symbol: ^
   - Example:
     - 0b1010 ^ 0b1100 = 0b0110

Bitwise operators are commonly used in computer programming for various tasks, including:

- Setting specific bits or clearing bits in a number.
- Extracting specific bits or ranges of bits from a number.
- Checking if a particular bit is set or cleared.
- Performing arithmetic operations on binary numbers efficiently.

These operators provide a low-level and efficient way to manipulate binary data and perform bitwise logic operations on individual bits of numbers. They are especially useful in embedded systems programming, cryptography, and low-level system programming.

10 **Why are the benefits for linked list over arrays.**

Linked lists offer several benefits over arrays, making them a preferred choice in certain scenarios:

1. Dynamic Size:
   - Linked lists can dynamically grow or shrink in size without the need for resizing or reallocation of memory. This flexibility allows for efficient memory usage and avoids the overhead of resizing operations.

2. Insertions and Deletions:
   - Insertions and deletions in a linked list are generally faster and more efficient than in arrays. In a singly linked list, inserting or deleting an element at the beginning or end of the list takes constant time (O(1)), while in an array, it requires shifting elements, resulting in a time complexity of O(n).

3. Memory Allocation:
   - Linked lists use dynamic memory allocation, which allows nodes to be allocated and deallocated as needed. This avoids the issue of memory fragmentation that can occur with arrays, especially when dealing with large or varying-sized data.

4. Contiguous Memory Allocation:
   - Unlike arrays, linked lists do not require contiguous memory allocation. This allows for efficient memory usage even when dealing with fragmented memory spaces.

5. Ease of Insertions in the Middle:
   - Insertions and deletions in the middle of a linked list can be performed efficiently by

adjusting pointers, without the need to shift or move other elements. This makes linked lists suitable for scenarios where frequent insertions or deletions are required.

6. Dynamic Data Structures:
   - Linked lists are versatile data structures that can be used to implement other abstract data types such as stacks, queues, and graphs. Their dynamic nature and efficient insertion/deletion operations make them suitable for implementing dynamic data structures.

7. Ease of Merging and Splitting:
   - Merging two linked lists or splitting a linked list into multiple parts can be done efficiently by adjusting pointers, without the need to copy elements. This makes linked lists suitable for certain types of algorithms and data manipulation tasks.

While linked lists offer these benefits, they also have some drawbacks compared to arrays, such as slower access time for random access and higher memory overhead due to storing pointers. Therefore, the choice between linked lists and arrays depends on the specific requirements and characteristics of the problem being solved.

11 **Implement singly linked list.**

```
class Node:
    def __init__(self, data):
        # Each node in the linked list contains some data and a reference to the next node.
        self.data = data
        self.next = None  # Initially, the next node is set to None.

class LinkedList:
    def __init__(self):
        # Initialize the linked list with a head node, which initially points to None.
        self.head = None

    def append(self, data):
        # Append a new node with the given data to the end of the linked list.
        new_node = Node(data)
        if self.head is None:
            # If the linked list is empty, set the new node as the head.
            self.head = new_node
            return
        # Traverse the list until the last node.
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        # Assign the new node to the next reference of the last node.
        last_node.next = new_node

    def prepend(self, data):
        # Add a new node with the given data to the beginning of the linked list.
        new_node = Node(data)
        new_node.next = self.head  # Make the new node point to the current head.
        self.head = new_node  # Update the head to the new node.

    def delete_node(self, data):
        # Delete the first occurrence of a node with the given data.
        current_node = self.head
        if current_node and current_node.data == data:
            # If the node to be deleted is the head node, update the head.
            self.head = current_node.next
            current_node = None  # Remove reference to the deleted node.
            return
        prev = None
        # Traverse the list to find the node to be deleted.
        while current_node and current_node.data != data:
            prev = current_node
            current_node = current_node.next
        if current_node is None:
            # If the node is not found, return.
            return
        # Remove the reference to the node to be deleted.
        prev.next = current_node.next
        current_node = None

    def display(self):
        # Display the elements of the linked list.
        current_node = self.head
        while current_node:
            print(current_node.data, end=' ')
            current_node = current_node.next
        print()

# Example usage:
if __name__ == "__main__":
    # Create a linked list
    linked_list = LinkedList()
    linked_list.append(1)
    linked_list.append(2)
    linked_list.append(3)
    linked_list.prepend(0)

    # Display the linked list
    linked_list.display()  # Output: 0 1 2 3

    # Delete a node
    linked_list.delete_node(2)

    # Display the linked list after deletion
    linked_list.display()  # Output: 0 1 3
```

11 Implement stack with singly linked list.

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

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

    def is_empty(self):
        return self.head is None
    def push(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    def pop(self):
        if self.is_empty():
            raise Exception("Stack is empty")
        data = self.head.data
        self.head = self.head.next
        return data

    def peek(self):
        if self.is_empty():
            raise Exception("Stack is empty")
        return self.head.data

    def display(self):
        current_node = self.head
        while current_node:
            print(current_node.data, end=' ')
            current_node = current_node.next
        print()

# Example usage:
if __name__ == "__main__":
    stack = Stack()

    # Push elements onto the stack
    stack.push(1)
    stack.push(2)
    stack.push(3)

    # Display the stack
    stack.display()  # Output: 3 2 1
    # Pop an element from the stack
    popped_element = stack.pop()
    print("Popped element:", popped_element)  # Output: Popped element: 3

    # Peek at the top element of the stack
    top_element = stack.peek()
    print("Top element:", top_element)  # Output: Top element: 2
```

12 **Implement queue with singly linked list.**

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

class Queue:
    def __init__(self):
        self.front = None  # Front of the queue
        self.rear = None   # Rear of the queue

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

    def enqueue(self, data):
        new_node = Node(data)
        if self.is_empty():
            self.front = new_node
            self.rear = new_node
        else:
            self.rear.next = new_node
            self.rear = new_node

    def dequeue(self):
        if self.is_empty():
            raise Exception("Queue is empty")
        data = self.front.data
        self.front = self.front.next
        if self.front is None:
            self.rear = None
        return data

    def peek(self):
        if self.is_empty():
            raise Exception("Queue is empty")
        return self.front.data

    def display(self):
        current_node = self.front
        while current_node:
            print(current_node.data, end=' ')
            current_node = current_node.next
        print()

# Example usage:
if __name__ == "__main__":
    queue = Queue()

    # Enqueue elements into the queue
    queue.enqueue(1)
    queue.enqueue(2)
    queue.enqueue(3)

    # Display the queue
    queue.display()  # Output: 1 2 3

    # Dequeue an element from the queue
    dequeued_element = queue.dequeue()
    print("Dequeued element:", dequeued_element)  # Output: Dequeued element: 1

    # Peek at the front element of the queue
    front_element = queue.peek()
    print("Front element:", front_element)  # Output: Front element: 2
```

12 **Implement doubly linked list.**

```
class Node:
    def __init__(self, data):
        # Each node in the doubly linked list contains some data,
        # and references to the previous and next nodes.
        self.data = data
        self.prev = None  # Reference to the previous node
        self.next = None  # Reference to the next node

class DoublyLinkedList:
    def __init__(self):
        self.head = None  # Initialize the head of the doubly linked list

    def append(self, data):
        # Append a new node with the given data to the end of the list.
        new_node = Node(data)
        if self.head is None:
            # If the list is empty, the new node becomes the head.
            self.head = new_node
            return
        # Traverse the list to find the last node.
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        # Set the next reference of the last node to the new node,
        # and set the previous reference of the new node to the last node.
        last_node.next = new_node
        new_node.prev = last_node

    def prepend(self, data):
        # Add a new node with the given data to the beginning of the list.
        new_node = Node(data)
        new_node.next = self.head  # Make the new node point to the current head.
        if self.head:
            self.head.prev = new_node  # Update the previous reference of the head.
        self.head = new_node  # Update the head to the new node.
    def delete(self, key):
        # Delete the first occurrence of a node with the specified data.
        current_node = self.head
        while current_node:
            if current_node.data == key:
                if current_node.prev:
                    # If the node to be deleted is not the first node,
                    # update the next reference of the previous node.
                    current_node.prev.next = current_node.next
                else:
                    # If the node to be deleted is the first node,
                    # update the head of the list.
                    self.head = current_node.next
                if current_node.next:
                    # If the node to be deleted is not the last node,
                    # update the previous reference of the next node.
                    current_node.next.prev = current_node.prev
                return
            current_node = current_node.next

    def display(self):
        # Display the elements of the doubly linked list.
        current_node = self.head
        while current_node:
            print(current_node.data, end=' ')
            current_node = current_node.next
        print()

# Example usage:
if __name__ == "__main__":
    # Create a doubly linked list
    dll = DoublyLinkedList()
    dll.append(1)
    dll.append(2)
    dll.append(3)
    dll.prepend(0)

    # Display the doubly linked list
    dll.display()  # Output: 0 1 2 3

    # Delete a node from the doubly linked list
    dll.delete(2)

    # Display the doubly linked list after deletion
    dll.display()  # Output: 0 1 3
```

13 **Implement circular linked list.**

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

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

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

    def display(self):
        if self.head is None:
            print("List is empty")
            return
        current = self.head
        while True:
            print(current.data, end=" ")
            current = current.next
            if current == self.head:
                break
        print()

# Example usage
cll = CircularLinkedList()
cll.append(1)
cll.append(2)
cll.append(3)
cll.display()  # Output: 1 2 3
```

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 = None
        self.rear = None

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

    def enqueue(self, data):
        new_node = Node(data)
        if self.is_empty():
            self.front = new_node
        else:
            self.rear.next = new_node
        self.rear = new_node
        self.rear.next = self.front  # Make rear's next point to front to make it circular

    def dequeue(self):
        if self.is_empty():
            raise Exception("Queue is empty")
        data = self.front.data
        if self.front == self.rear:
            self.front = None
            self.rear = None
        else:
            self.front = self.front.next
            self.rear.next = self.front  # Update rear's next to point to the new front
        return data

    def display(self):
        if self.is_empty():
            print("Queue is empty")
            return
        current_node = self.front
        while True:
            print(current_node.data, end=' ')
            current_node = current_node.next
            if current_node == self.front:
                break
        print()

# Example usage:
if __name__ == "__main__":
    queue = CircularQueue()

    # Enqueue elements into the circular queue
    queue.enqueue(1)
    queue.enqueue(2)
    queue.enqueue(3)
```

```
# Display the circular queue
queue.display()  # Output: 1 2 3

# Dequeue an element from the circular queue
dequeued_element = queue.dequeue()
print("Dequeued element:", dequeued_element)  # Output: Dequeued element: 1

# Display the circular queue after dequeuing
queue.display()  # Output: 2 3

# Enqueue another element
queue.enqueue(4)

# Display the circular queue after enqueuing
queue.display()  # Output: 2 3 4
```

15 **What is recursion? What is tail recursion?**

Recursion is a programming technique where a function calls itself directly or indirectly in order to solve a problem. It's a powerful concept that allows solutions to be expressed in a concise and elegant manner. Recursion involves breaking down a problem into smaller, similar subproblems and solving each subproblem recursively until a base case is reached.

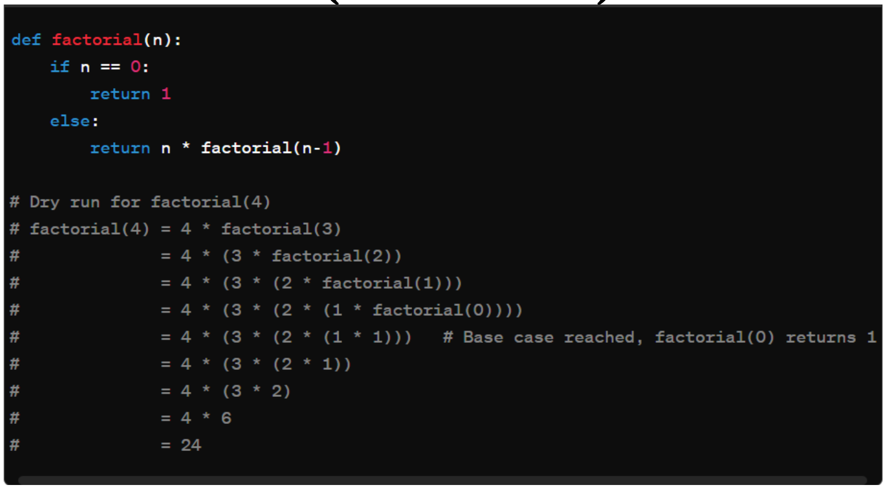
A recursive function typically consists of two parts:

1. Base Case(s): These are the terminating conditions that stop the recursion. When the base case is reached, the function stops calling itself and returns a result without further recursion.

2. Recursive Case(s): These are the conditions where the function calls itself with modified arguments to solve smaller instances of the same problem. The function continues to call itself recursively until it reaches the base case.

For example, consider the factorial function:

Advanced Algorithmic Problem Solving (R1UC601B)



```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

# Dry run for factorial(4)
# factorial(4) = 4 * factorial(3)
#              = 4 * (3 * factorial(2))
#              = 4 * (3 * (2 * factorial(1)))
#              = 4 * (3 * (2 * (1 * factorial(0))))
#              = 4 * (3 * (2 * (1 * 1)))   # Base case reached, factorial(0) returns 1
#              = 4 * (3 * (2 * 1))
#              = 4 * (3 * 2)
#              = 4 * 6
#              = 24
```

Here, the base case is when `n` equals 0, and the recursive case is when `n` is greater than 0.

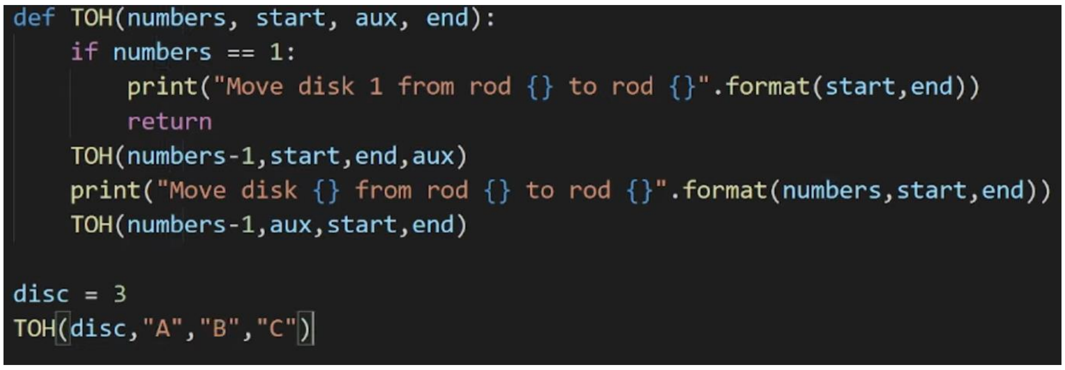
Tail recursion is a special form of recursion where the recursive call is the last operation performed by the function. In other words, the result of the recursive call is immediately returned without further computation. Tail recursion is often optimized by compilers or interpreters to avoid stack overflow errors and improve performance.

```
def factorial_tail(n, result=1):
    if n == 0:
        return result
    else:
        return factorial_tail(n-1, n*result)


# Dry run for factorial_tail(4)
# factorial_tail(4, 1) = factorial_tail(3, 4*1)
#                      = factorial_tail(2, 3*4*1)
#                      = factorial_tail(1, 2*3*4*1)
#                      = factorial_tail(0, 1*2*3*4*1)
#                      = 1*2*3*4*1        # Base case reached, factorial_tail(0, result) returns
result
#                      = 24
```

For example, consider the tail-recursive version of the factorial function:

In this tail-recursive implementation, the recursive call (`factorial_tail(n-1, n*result)`) is the last operation performed before returning the result. This allows for efficient tail call optimization by some programming languages or compilers, where each recursive call doesn't need to create a new stack frame, saving memory and potentially improving performance.

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.**



```
def TOH(numbers, start, aux, end):
    if numbers == 1:
        print("Move disk 1 from rod {} to rod {}".format(start,end))
        return
    TOH(numbers-1,start,end,aux)
    print("Move disk {} from rod {} to rod {}".format(numbers,start,end))
    TOH(numbers-1,aux,start,end)

disc = 3
TOH(disc,"A","B","C")
```

- Time Complexity: The time complexity of the Tower of Hanoi problem is O(2^n), where n is the number of disks. This is because each disk has to be moved 2^n - 1 times to complete the puzzle. The time complexity arises from the recursive nature of the algorithm.
- Space Complexity: The space complexity of the Tower of Hanoi problem is O(n), where n is the number of disks. This is because the recursion stack can grow up to n levels deep during the execution of the algorithm. Additionally, each function call requires O(1) space for storing its local variables and parameters.

17 **What is backtracking in algorithms? What kind of problems are solved with this technique?**

Backtracking is a powerful algorithmic technique used to systematically search for solutions to combinatorial problems, such as constraint satisfaction problems, permutation problems, and optimization problems. It is especially useful for problems where the solution needs to be built incrementally, and it's often used when exhaustive search is impractical due to the large search space.

The basic idea behind backtracking is to recursively explore all possible solutions to a problem by making a series of choices, and if a choice leads to a dead end (i.e., violates a constraint or cannot lead to a solution), the algorithm backtracks to the previous decision point and tries another alternative.

Here are some key characteristics of backtracking:

1. Recursive approach: Backtracking typically involves recursion, where the search space is explored depth-first.
2. State space search: The algorithm explores the state space of the problem by making choices and exploring each choice until a solution is found or all possibilities are exhausted.
3. Pruning: Backtracking often involves pruning the search space by eliminating branches that cannot lead to a valid solution. This helps improve efficiency by avoiding unnecessary exploration.
4. Incremental construction: Solutions are built incrementally, and if a partial solution cannot be extended to a complete solution, the algorithm backtracks and tries another alternative.

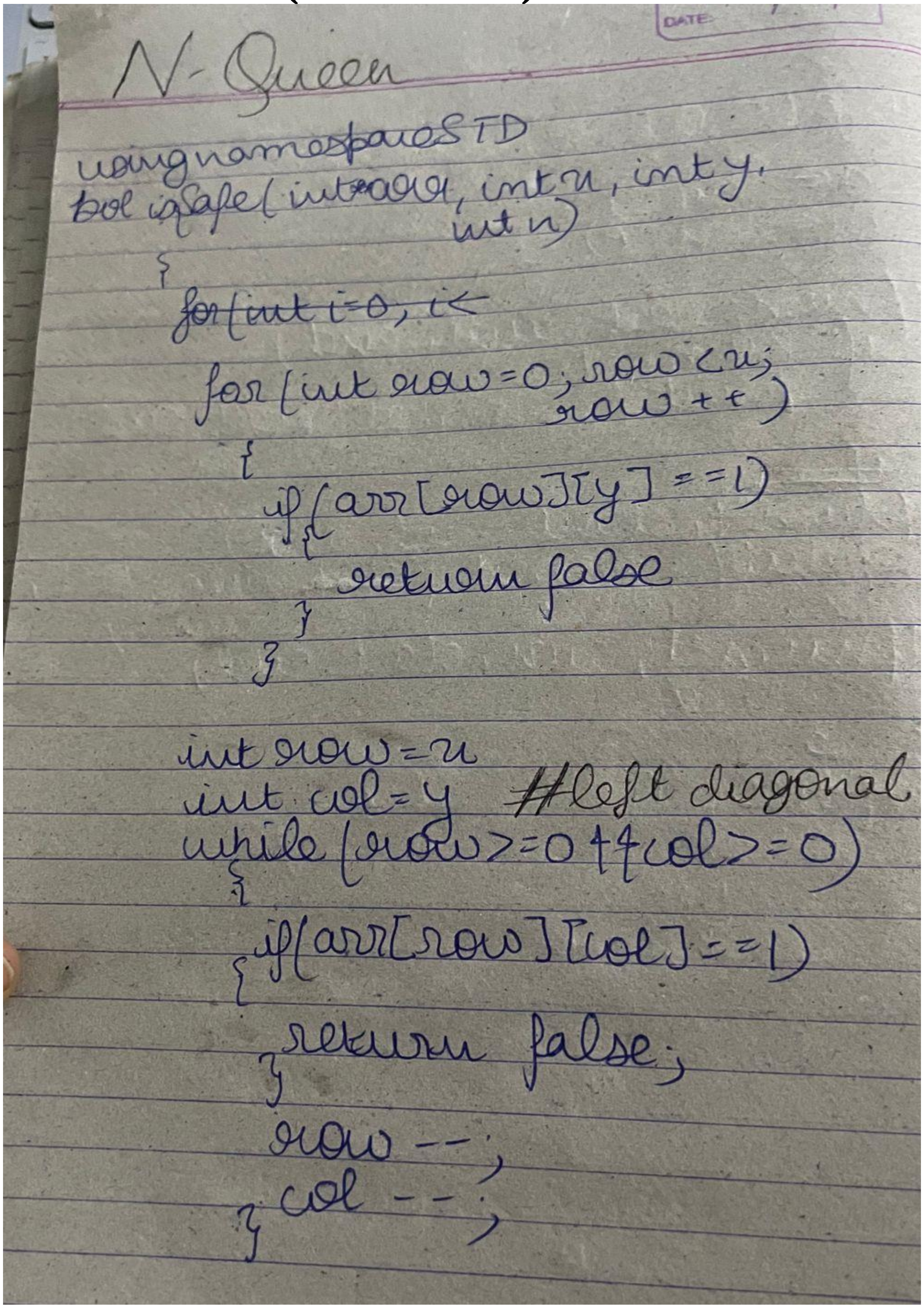
Backtracking is commonly used to solve problems such as:

- N-Queens Problem: Placing N queens on an NxN chessboard such that no two queens threaten each other.
- Subset Sum Problem: Finding a subset of a given set of integers that adds up to a given sum.
- Sudoku: Filling in a 9x9 grid with digits 1 through 9 such that each column, row, and 3x3 subgrid contains all of the digits from 1 to 9 without repetition.
- Hamiltonian Cycle Problem: Finding a cycle that visits every vertex in a graph exactly once.
- Graph Coloring Problem: Coloring the vertices of a graph such that no two adjacent vertices have the same color, using the fewest possible colors.

Backtracking is a fundamental technique in algorithm design and is used in various real-world applications, such as scheduling, resource allocation, and puzzle solving.

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

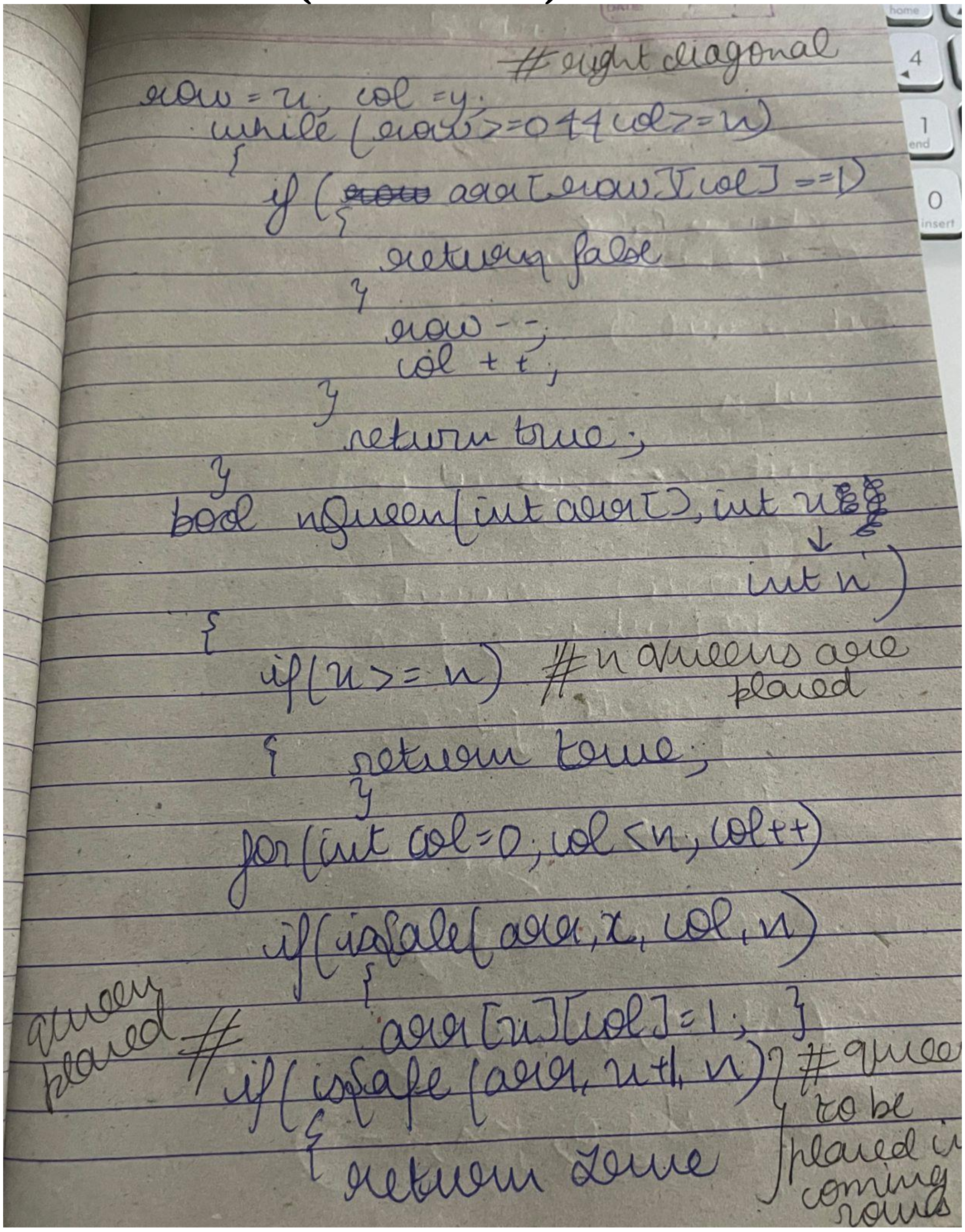
Advanced Algorithmic Problem Solving (R1UC601B)



N-Queen

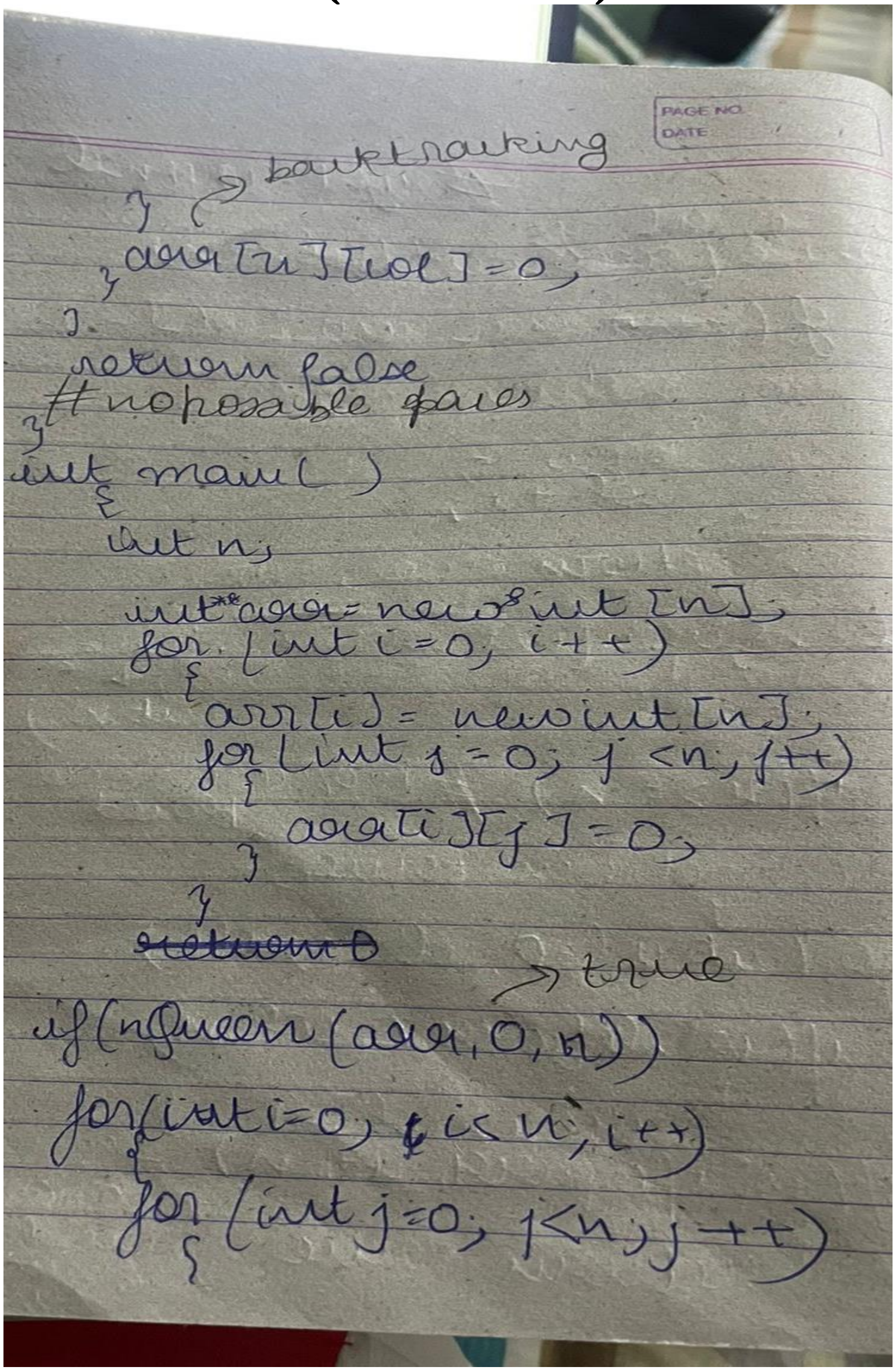
```
using namespace STD
bool isSafe( int arr, int x, int y,
                         int n)
    {
     ~~for(int i=0, i<~~
      for (int row=0; row<x;
                          row++)
       {
        if(arr[row][y]==1)
        {
         return false
        }
       }

      int row=x
      int col=y    #left diagonal
      while (row>=0 && col>=0)
       {
        if(arr[row][col]==1)
        {
         return false;
        }
        row--;
        col--;
       }
```

Advanced Algorithmic Problem Solving (R1UC601B)



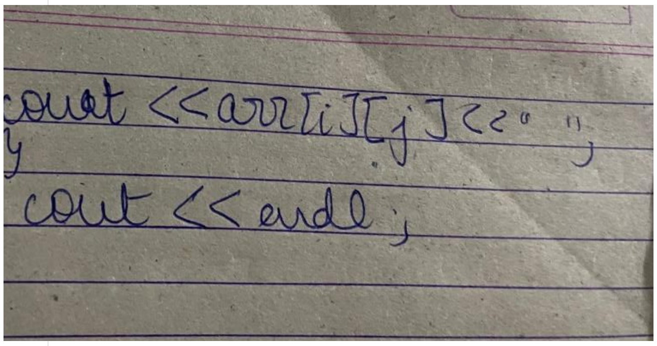
```
                        # right diagonal
row = x. col = y.
 while (row>=0 && col>=n)
  {
   if (~~row~~ arr[row][col] ==1)
     {
       return false
      }
       row--;
       col++;
    }
     return true;
 }
bool nQueen(int arr[], int x
                            ↓
                          int n)
 {
   if(x>=n)  # n queens are
                    placed
   {  return true;
    }
   for(int col=0; col<n; col++)
    if(isSafe(arr, x, col, n)
       {
        arr[x][col]=1;  }
queen
placed #
   if( isSafe (arr, x+1, n)) # queen
    {                          to be
     return true               placed in
                               coming
                               rows
```

Advanced Algorithmic Problem Solving (R1UC601B)



PAGE NO.
DATE

```
        } ↗ backtracking
        }
      arr[r][col] = 0;
    }
   }
 return false
 #nopossible spaces
}
int main( )
{
   int n;
   int**arr= new*int [n];
   for (int i=0, i++)
     {
      arr[i] = newint[n];
      for(int j=0; j<n; j++)
       {
        arr[i][j] = 0;
       }
     }
   ~~return 0~~
                  → true
 if(nqueen(arr,0,n))
 for(int i=0; i<n; i++)
 {
  for (int j=0; j<n; j++)
  {
```

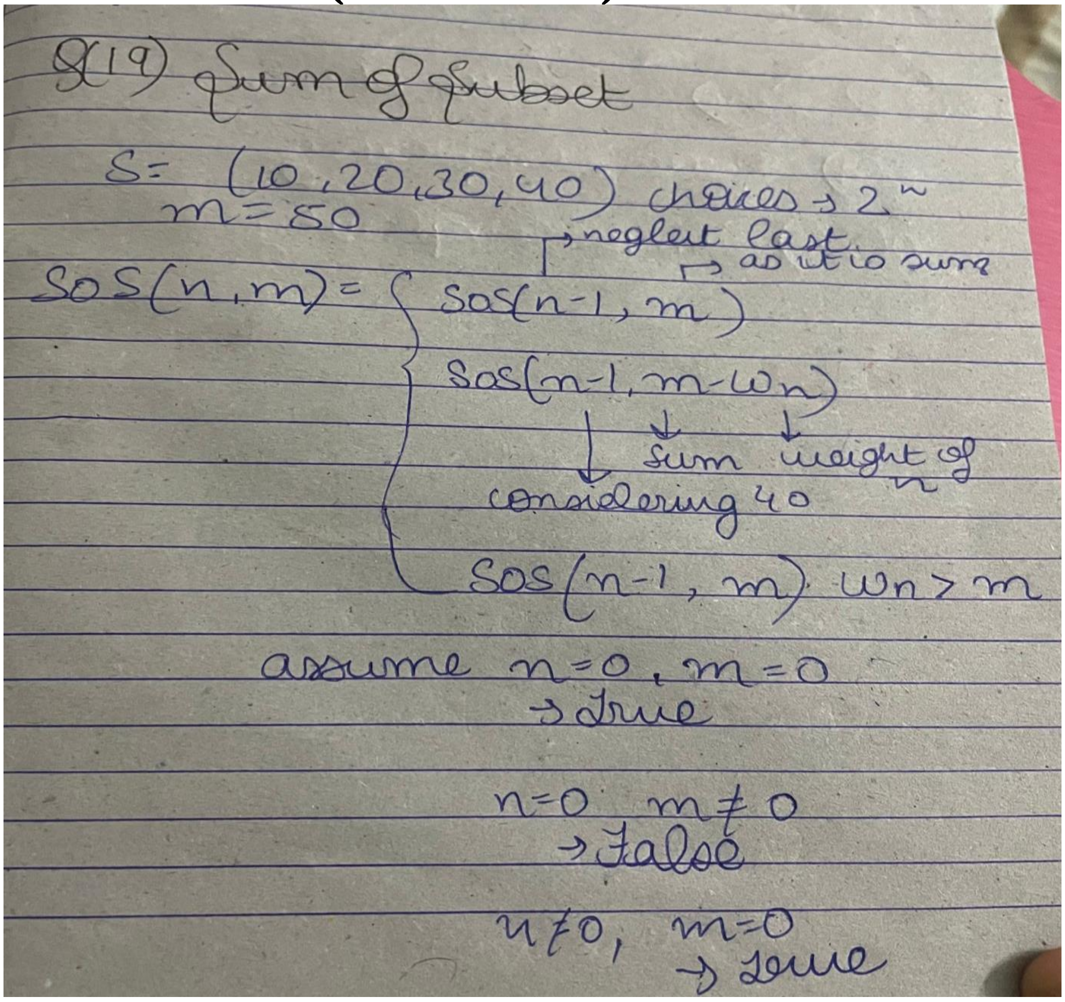


```
cout <<arr[i][j]<<" ";
}
cout <<endl;
```

-
- Time Complexity: The time complexity of this program is exponential, typically O(N!), where N is the number of queens. This is because the program tries all possible combinations of queen placements on the NxN chessboard, and the number of such combinations grows rapidly as N increases.
- Space Complexity: The space complexity of this program is O(N^2), where N is the number of queens. This is because we use a 2D array (`board`) to represent the chessboard, which requires O(N^2) space. Additionally, the recursive call stack can grow up to N levels deep during the execution of the backtracking algorithm.

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

Advanced Algorithmic Problem Solving (R1UC601B)



Q(19) Sum of Subset

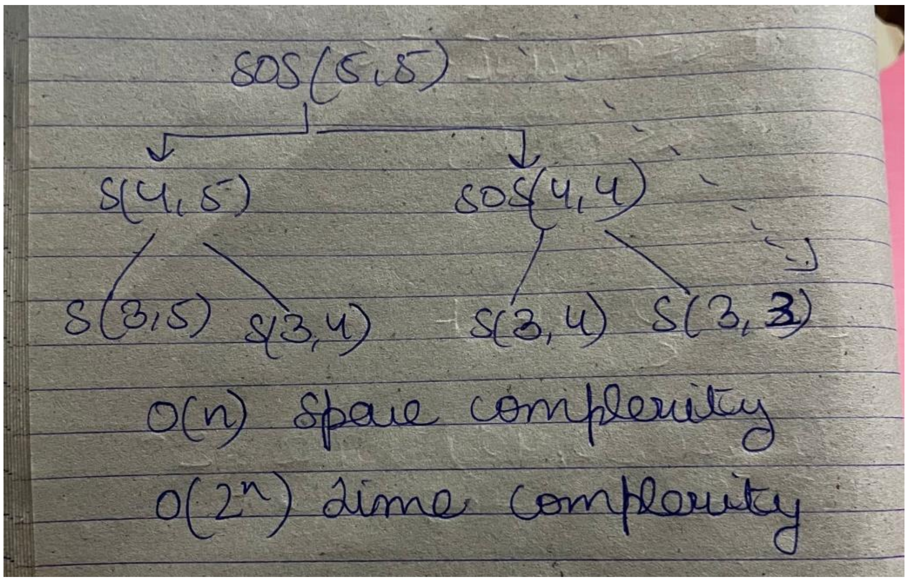
S = (10, 20, 30, 40) choices → 2^n
m = 50

$$SOS(n,m) = \begin{cases} SOS(n-1, m) & \text{→ neglect last → as it is sum} \\ SOS(n-1, m-w_n) & \text{considering 40; } m \text{: sum, } w_n \text{: weight of } n \\ SOS(n-1, m) & w_n > m \end{cases}$$

assume n = 0, m = 0
→ True

n = 0 m ≠ 0
→ False

n ≠ 0, m = 0
→ True

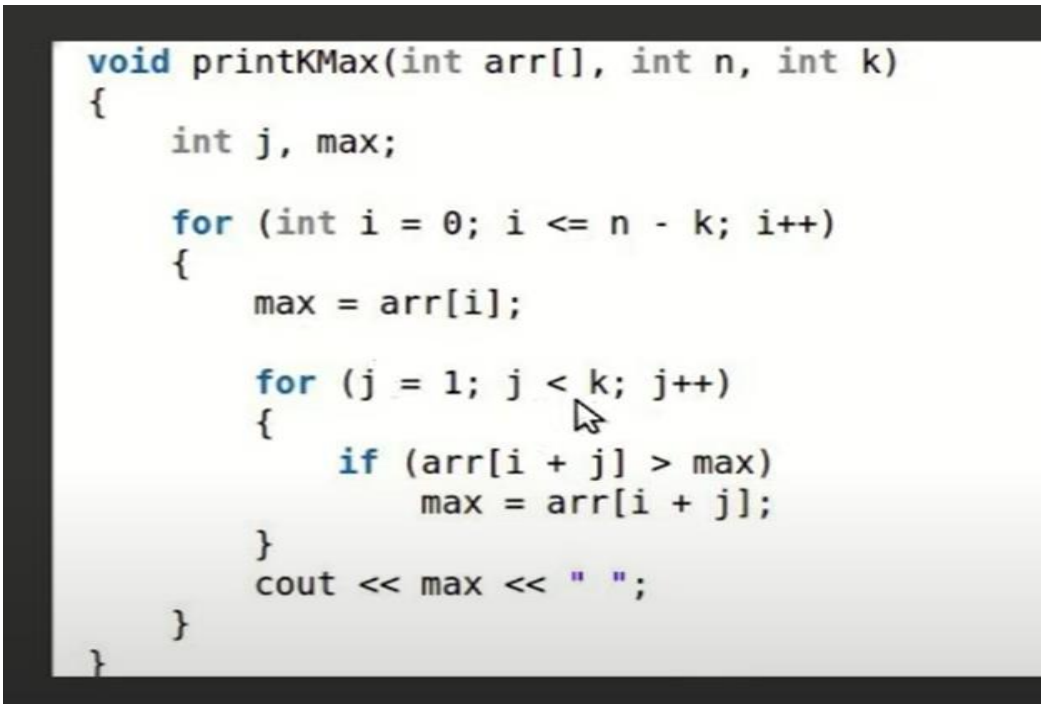


$$\begin{tikzcd}[column sep=small] & & SOS(5,5) \arrow[dll] \arrow[drr] & & \\ S(4,5) \arrow[d] \arrow[dr] & & & & SOS(4,4) \arrow[d] \arrow[dr] \\ S(3,5) & S(3,4) & & & S(3,4) & S(3,3) \end{tikzcd}$$

O(n) Space complexity
O(2^n) time complexity

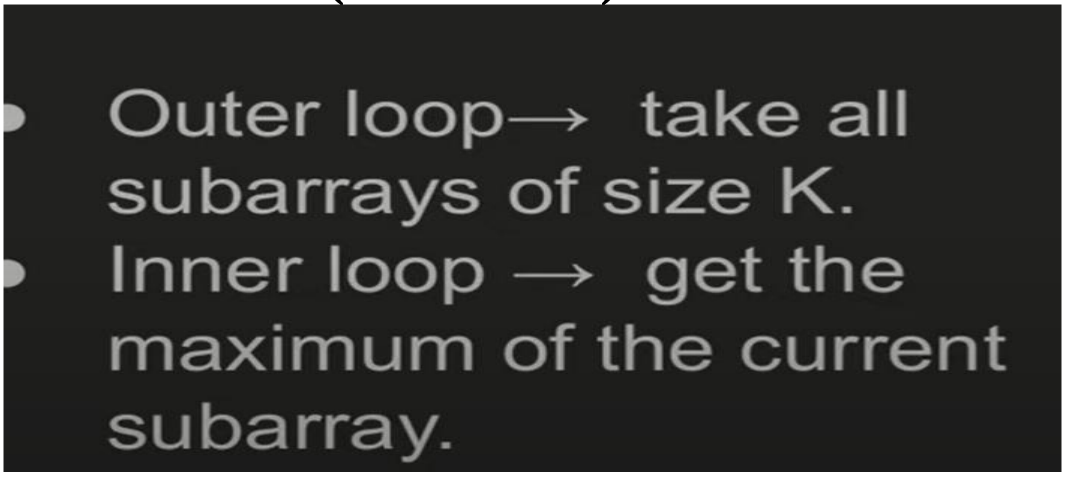
```
def subset_sum(nums, target, n):
    # Base cases
    if target == 0:
        return True
    if n == 0:
        return False

    # If the last element is greater than the target, it can't be included
    if nums[n-1] > target:
        return subset_sum(nums, target, n-1)

    # Recursive call to check if the target can be obtained by including or excluding the
last element
    return subset_sum(nums, target - nums[n-1], n-1) or subset_sum(nums, target, n-1)

# Example usage
nums = [3, 34, 4, 12, 5, 2]
target = 9
n = len(nums)
if subset_sum(nums, target, n):
    print("Subset with the given sum exists")
else:
    print("Subset with the given sum does not exist")
```

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



```
void printKMax(int arr[], int n, int k)
{
    int j, max;

    for (int i = 0; i <= n - k; i++)
    {
        max = arr[i];

        for (j = 1; j < k; j++)
        {
            if (arr[i + j] > max)
                max = arr[i + j];
        }
        cout << max << " ";
    }
}
```

Advanced Algorithmic Problem Solving (R1UC601B)



- Outer loop→ take all subarrays of size K.
- Inner loop → get the maximum of the current subarray.

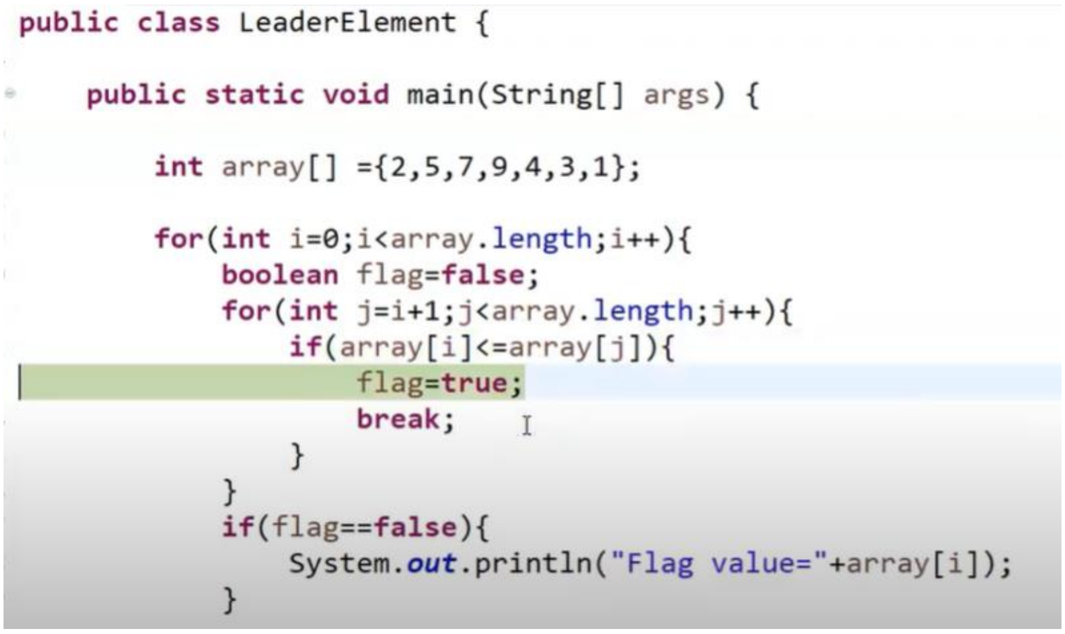
21 **Write a recursive function to generate all possible subsets of a given set.**

Same as 19

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

Same as 29

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.**



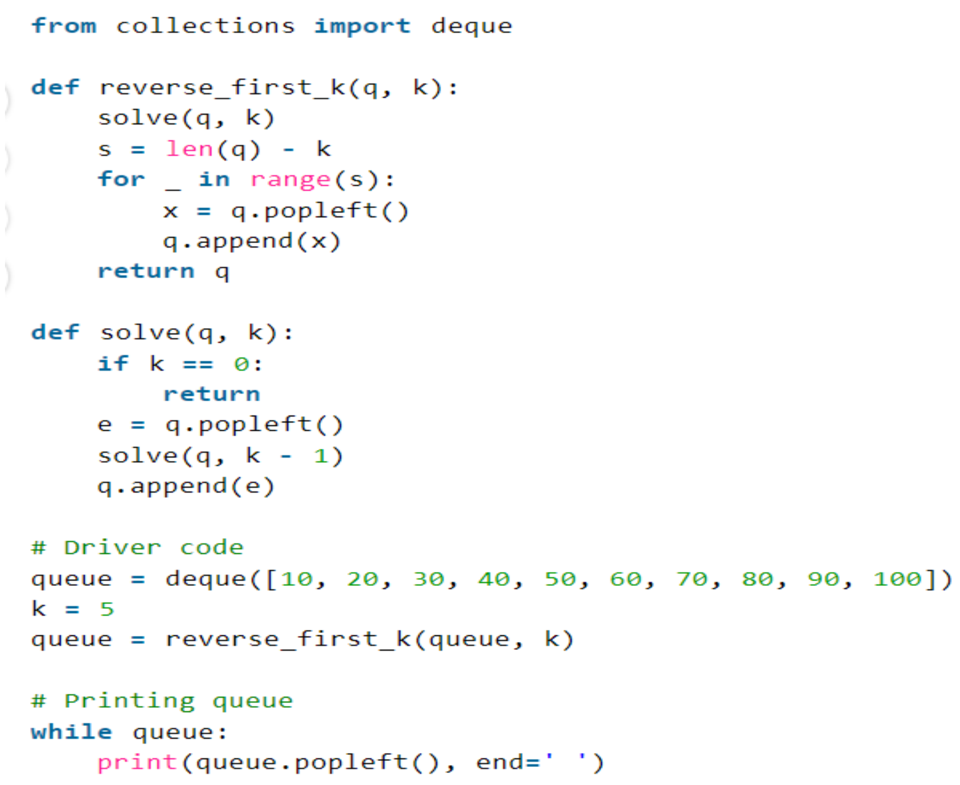
```
public class LeaderElement {

    public static void main(String[] args) {

        int array[] ={2,5,7,9,4,3,1};

        for(int i=0;i<array.length;i++){
            boolean flag=false;
            for(int j=i+1;j<array.length;j++){
                if(array[i]<=array[j]){
                    flag=true;
                    break;
                }
            }
            if(flag==false){
                System.out.println("Flag value="+array[i]);
            }
```

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.**

Same as 29

Advanced Algorithmic Problem Solving (R1UC601B)

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.**



```python
from collections import deque

def reverse_first_k(q, k):
    solve(q, k)
    s = len(q) - k
    for _ in range(s):
        x = q.popleft()
        q.append(x)
    return q

def solve(q, k):
    if k == 0:
        return
    e = q.popleft()
    solve(q, k - 1)
    q.append(e)

# Driver code
queue = deque([10, 20, 30, 40, 50, 60, 70, 80, 90, 100])
k = 5
queue = reverse_first_k(queue, k)

# Printing queue
while queue:
    print(queue.popleft(), end=' ')
```

26 **Write a program to implement a stack using queues.**

27 **Wrire a program to implement queue using stacks.**

```python
class QueueUsingStacks:
    def __init__(self):
        self.enqueue_stack = []
        self.dequeue_stack = []


    def enqueue(self, val):
        # Push the element onto the enqueue stack
        self.enqueue_stack.append(val)


    def dequeue(self):
        if not self.dequeue_stack:
            # If the dequeue stack is empty, transfer elements from the enqueue stack
            while self.enqueue_stack:
                self.dequeue_stack.append(self.enqueue_stack.pop())
        # Pop the top element from the dequeue stack
        if self.dequeue_stack:
            return self.dequeue_stack.pop()
        else:
            return None

    def is_empty(self):
        return not (self.enqueue_stack or self.dequeue_stack)

# Example usage:
queue = QueueUsingStacks()
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)

print("Dequeue:", queue.dequeue())  # Output: 1
print("Dequeue:", queue.dequeue())  # Output: 2

queue.enqueue(4)

print("Dequeue:", queue.dequeue())  # Output: 3
print("Dequeue:", queue.dequeue())  # Output: 4
print("Dequeue:", queue.dequeue())  # Output: None (Queue is empty)
```

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(s):
    seen = {}
    for char in s:
        if char in seen:
            return False
        seen[char] = True
    return True

# Example usage:
string1 = "hello"
string2 = "world"
```

```
print(string1, "is an isogram:", is_isogram(string1))  # Output: False
print(string2, "is an isogram:", is_isogram(string2))  # Output: True
```

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

```
def find_frequencies(arr):
    frequencies = {}
    n = len(arr)

    # Initialize the first element's frequency
    frequencies[arr[0]] = 1
    /
    # Iterate through the array to count frequencies
    for i in range(1, n):
        if arr[i] == arr[i - 1]:
            # If the current element is the same as the previous one,
            # increment its frequency
            frequencies[arr[i]] += 1
        else:
            # If the current element is different from the previous one,
            # initialize its frequency to 1
            frequencies[arr[i]] = 1

    return frequencies

# Example usage:
arr = [1, 1, 2, 2, 2, 3, 4, 4, 5, 5, 5, 5]
print("Original array:", arr)
print("Frequencies of each element:")
frequencies = find_frequencies(arr)
for key, value in frequencies.items():
    print(key, "->", value)
```

30 **Write a program to delete middle element from stack.**

```
class Stack:
    def __init__(self):
        self.stack = []
    def push(self, val):
        self.stack.append(val)

    def pop(self):
        if self.is_empty():
            return None
        return self.stack.pop()

    def is_empty(self):
        return len(self.stack) == 0

    def size(self):
        return len(self.stack)

    def delete_middle(self):
        if self.is_empty():
            return

        mid = self.size() // 2
        if self.size() % 2 == 0:  # Even number of elements
            # Remove the second middle element
            self.stack.pop(mid)
        else:  # Odd number of elements
            # Remove the middle element
            self.stack.pop(mid)

    def print_stack(self):
        print("Stack:", self.stack)

# Example usage:
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)
stack.push(4)
```

stack.push(5)

print("Original stack:")

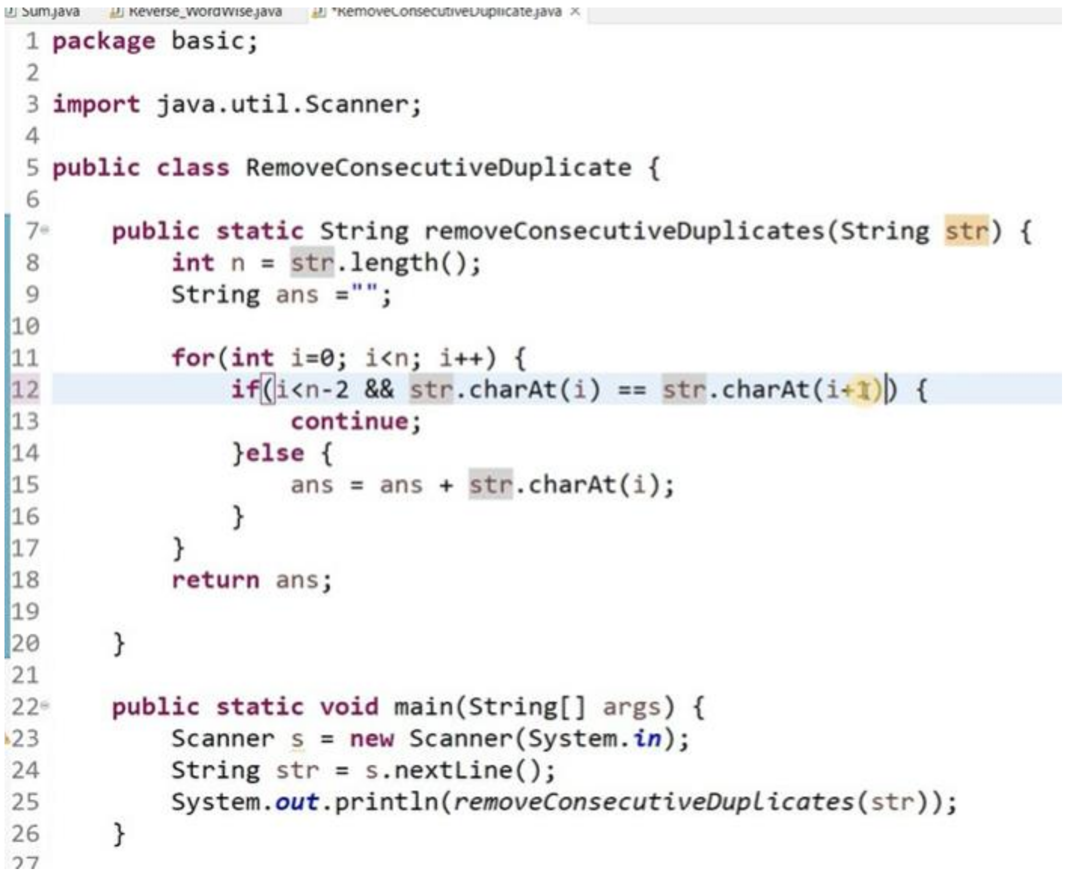
stack.print_stack()

stack.delete_middle()

print("Stack after deleting middle element:")

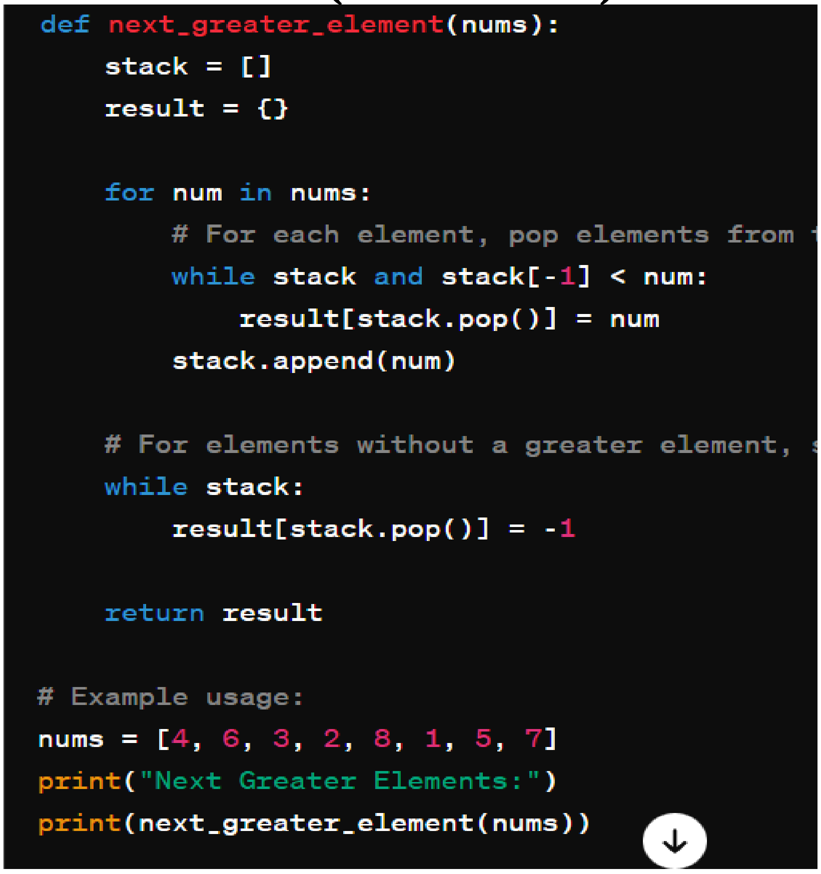
stack.print_stack()

31 **Write a program to remove consecutive duplicates from string.**



```java
package basic;

import java.util.Scanner;

public class RemoveConsecutiveDuplicate {

    public static String removeConsecutiveDuplicates(String str) {
        int n = str.length();
        String ans ="";

        for(int i=0; i<n; i++) {
            if(i<n-2 && str.charAt(i) == str.charAt(i+1)) {
                continue;
            }else {
                ans = ans + str.charAt(i);
            }
        }
        return ans;

    }

    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        String str = s.nextLine();
        System.out.println(removeConsecutiveDuplicates(str));
    }
```

33 **Write a program to display next greater element of all element given in array.**



```python
def next_greater_element(nums):
    stack = []
    result = {}

    for num in nums:
        # For each element, pop elements from t
        while stack and stack[-1] < num:
            result[stack.pop()] = num
        stack.append(num)

    # For elements without a greater element, s
    while stack:
        result[stack.pop()] = -1

    return result

# Example usage:
nums = [4, 6, 3, 2, 8, 1, 5, 7]
print("Next Greater Elements:")
print(next_greater_element(nums))
```

34 **Write a program to evaluate a postfix expression.**

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

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

            if token == '+':
                stack.append(operand1 + operand2)
            elif token == '-':
                stack.append(operand1 - operand2)
            elif token == '':
                stack.append(operand1  operand2)
            elif token == '/':
```

```
            stack.append(operand1 // operand2)  # Integer division to handle division by
zero

    return stack[0]

# Example usage:
postfix_expression = "4 5 7  + 2 -"
print("Result of evaluating postfix expression:", evaluate_postfix(postfix_expression))  #
Output: 39
```

35 **Write a program to get MIN at pop from stack.**

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

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

    def pop(self):
        if not self.stack:
            return None
        val = self.stack.pop()
        if val == self.min_stack[-1]:
            self.min_stack.pop()
        return val

    def get_min(self):
        if not self.min_stack:
            return None
        return self.min_stack[-1]

# Example usage:
stack = MinStack()
stack.push(3)
stack.push(5)
stack.push(2)
stack.push(1)

print("Minimum element at pop:", stack.get_min())  # Output: 1
print("Popped element:", stack.pop())           # Output: 1
print("Minimum element at pop:", stack.get_min())  # Output: 2
```

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

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def get_length(head):
    length = 0
    current = head
    while current:
        length += 1
        current = current.next
    return length

def swap_kth_node(head, k):
    length = get_length(head)

    # Check if k is out of bounds
    if k < 1 or k > length:
        return head

    # Find the k-th node from the beginning
    prev_kth_from_beginning = None
    current = head
    for _ in range(k - 1):
        prev_kth_from_beginning = current
        current = current.next

    # Find the k-th node from the end
    prev_kth_from_end = None
    kth_from_end = head
    for _ in range(length - k):
        prev_kth_from_end = kth_from_end
        kth_from_end = kth_from_end.next

    # Swap values of k-th nodes
    if prev_kth_from_beginning:
        prev_kth_from_beginning.next, kth_from_end.next = kth_from_end.next,
prev_kth_from_beginning.next
    else:
        head, kth_from_end.next = kth_from_end.next, head

    if prev_kth_from_end:
        prev_kth_from_end.next, current.next = current.next, prev_kth_from_end.next
    else:
        head, current.next = current.next, head

    return head

def print_list(head):
    temp = head
    while temp:
        print(temp.val, end=" ")
        temp = temp.next
    print()

# Example usage:
# Create a linked list: 1 -> 2 -> 3 -> 4 -> 5 -> 6
head = ListNode(1)
head.next = ListNode(2)
head.next.next = ListNode(3)
head.next.next.next = ListNode(4)
head.next.next.next.next = ListNode(5)
head.next.next.next.next.next = ListNode(6)

print("Original List:")
print_list(head)

k = 2
head = swap_kth_node(head, k)
```

```
print(f"After swapping {k}-th node from ends:")
print_list(head)
```

37 **Write a program to detect loop in linked list**

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def has_cycle(head):
    if not head or not head.next:
        return False

    slow = head
    fast = head.next

    # Move slow pointer by one step and fast pointer by two steps
    # If there is a loop, they will eventually meet
    while fast and fast.next:
        if slow == fast:
            return True
        slow = slow.next
        fast = fast.next.next

    return False

# Example usage:
# Create a linked list with a cycle: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 3 (back to node 3)
head = ListNode(1)
head.next = ListNode(2)
head.next.next = ListNode(3)
head.next.next.next = ListNode(4)
head.next.next.next.next = ListNode(5)
head.next.next.next.next.next = ListNode(6)
head.next.next.next.next.next.next = head.next.next  # Create a cycle

if has_cycle(head):
    print("Linked list has a cycle.")
else:
    print("Linked list does not have a cycle.")
```

38 **Write a program to find Intersection point in Y shaped Linked list.**

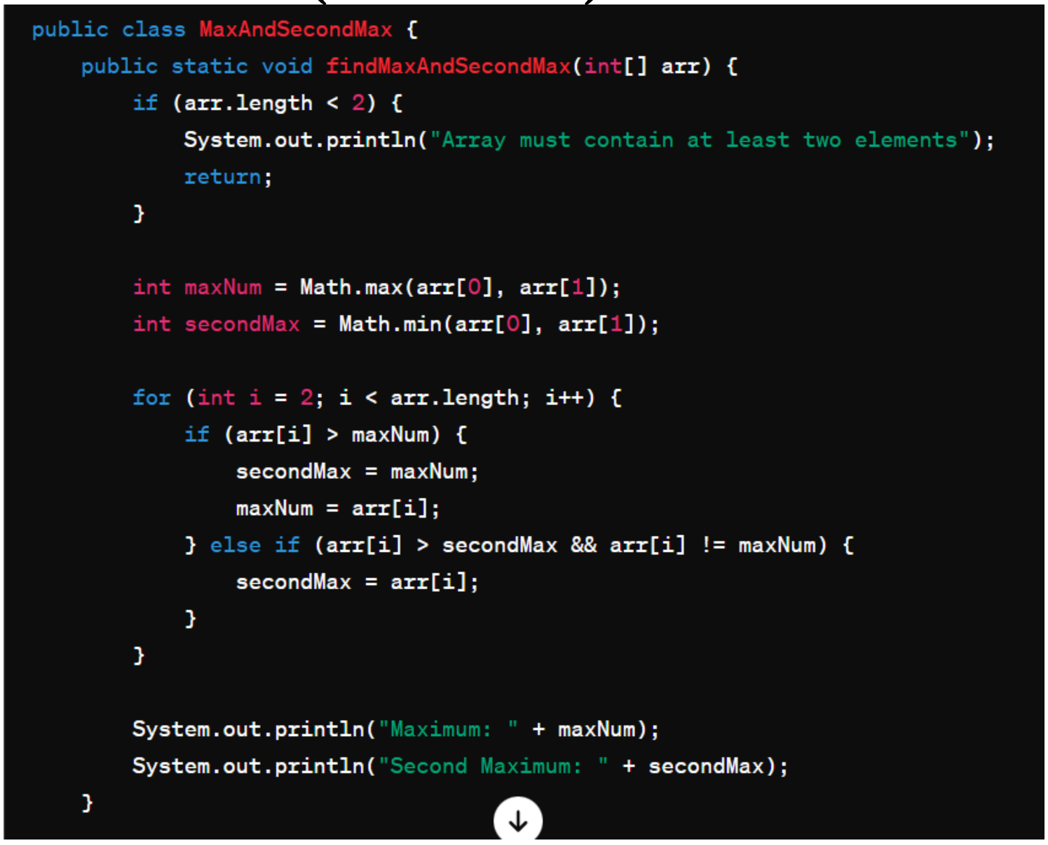
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

def find_intersection(head1, head2):
    # Calculate the lengths of the two linked lists.
    length1 = 0
    temp1 = head1
    while temp1:
        length1 += 1
        temp1 = temp1.next

    length2 = 0
    temp2 = head2
    while temp2:
        length2 += 1
        temp2 = temp2.next

    # Move the pointer of the longer linked list to the beginning of the shorter linked list.
    if length1 > length2:
        for i in range(length1 - length2):
            head1 = head1.next
    elif length2 > length1:
        for i in range(length2 - length1):
            head2 = head2.next

    # Traverse the two linked lists simultaneously and compare the nodes.
    while head1 and head2:
        if head1 == head2:
            return head1
        head1 = head1.next
        head2 = head2.next

    # If no intersection point is found, return None.
    return None
```

```
# Create the two linked lists.
head1 = Node(1)
head1.next = Node(2)
head1.next.next = Node(3)
head1.next.next.next = Node(4)
head1.next.next.next.next = Node(5)

head2 = Node(6)
head2.next = Node(7)
head2.next.next = Node(8)
head2.next.next.next = head1.next.next.next

# Find the intersection point.
intersection_point = find_intersection(head1, head2)

# Print the intersection point.
if intersection_point:
    print("The intersection point is:", intersection_point.data)
else:
    print("No intersection point found.")
```

39 **Write a program to merge two sorted linked list.**

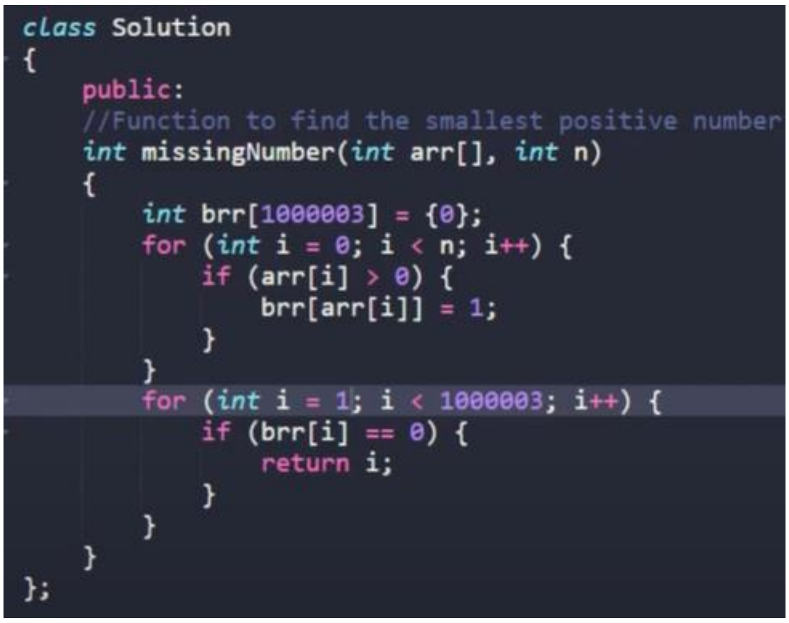
```
def merge_sorted(head1, head2):
  # if both lists are empty then merged list is also empty
  # if one of the lists is empty then other is the merged list
  if head1 == None:
    return head2
  elif head2 == None:
    return head1

  mergedHead = None;
  if head1.data <= head2.data:
    mergedHead = head1
    head1 = head1.next
  else:
    mergedHead = head2
    head2 = head2.next

  mergedTail = mergedHead
  while head1 != None and head2 != None:
    temp = None
    if head1.data <= head2.data:
      temp = head1
      head1 = head1.next
    else:
      temp = head2
      head2 = head2.next

    mergedTail.next = temp
    mergedTail = temp

  if head1 != None:
    mergedTail.next = head1
  elif head2 != None:
    mergedTail.next = head2

  return mergedHead



array1 = [2, 3, 5, 6]
array2 = [1, 4, 10]
list_head1 = create_linked_list(array1)
print("Original1:")
display (list_head1)
list_head2 = create_linked_list(array2)
print("\nOriginal2:")
display (list_head2)
new_head = merge_sorted(list_head1, list_head2)

print("\nMerged:")
  display(new_head)
```

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

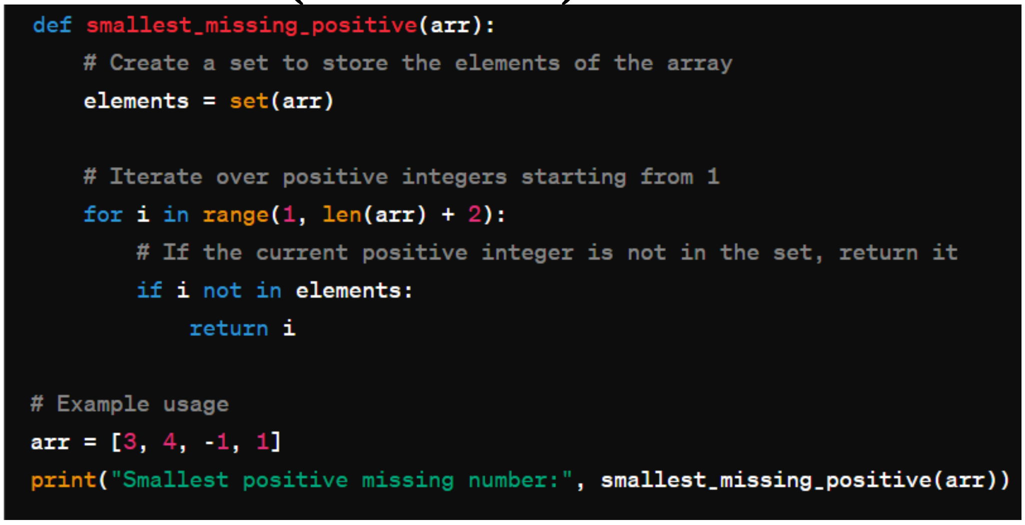


```java
public class MaxAndSecondMax {
    public static void findMaxAndSecondMax(int[] arr) {
        if (arr.length < 2) {
            System.out.println("Array must contain at least two elements");
            return;
        }

        int maxNum = Math.max(arr[0], arr[1]);
        int secondMax = Math.min(arr[0], arr[1]);

        for (int i = 2; i < arr.length; i++) {
            if (arr[i] > maxNum) {
                secondMax = maxNum;
                maxNum = arr[i];
            } else if (arr[i] > secondMax && arr[i] != maxNum) {
                secondMax = arr[i];
            }
        }

        System.out.println("Maximum: " + maxNum);
        System.out.println("Second Maximum: " + secondMax);
    }
```

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.**

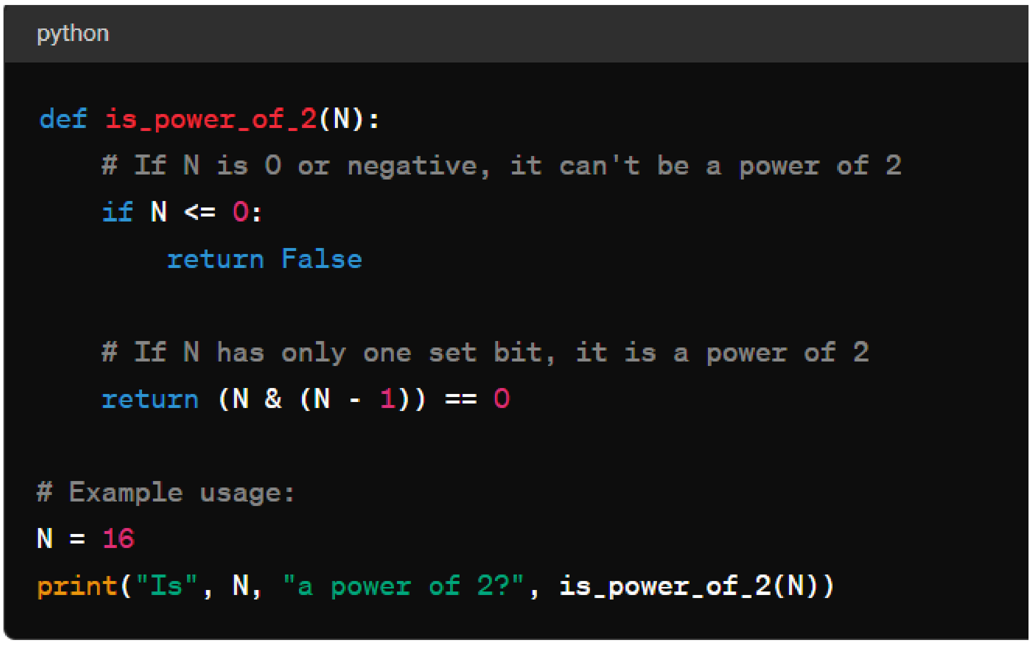


```cpp
class Solution
{
    public:
    //Function to find the smallest positive number
    int missingNumber(int arr[], int n)
    {
        int brr[1000003] = {0};
        for (int i = 0; i < n; i++) {
            if (arr[i] > 0) {
                brr[arr[i]] = 1;
            }
        }
        for (int i = 1; i < 1000003; i++) {
            if (brr[i] == 0) {
                return i;
            }
        }
    }
};
```



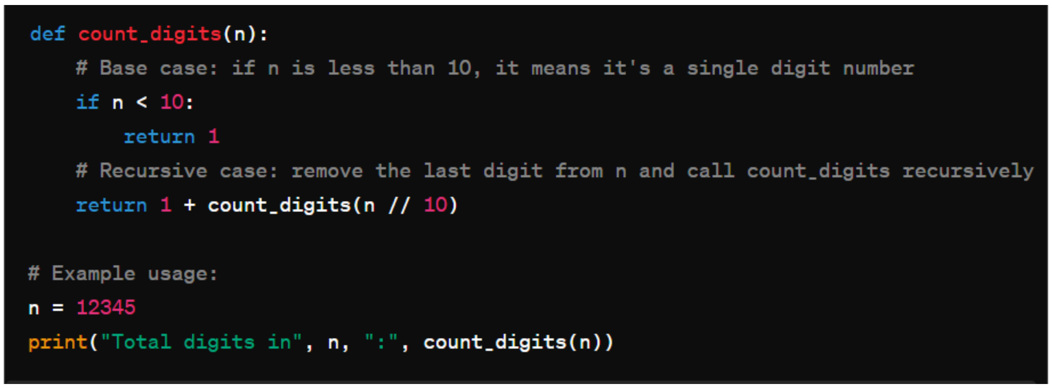
```python
def smallest_missing_positive(arr):
    # Create a set to store the elements of the array
    elements = set(arr)

    # Iterate over positive integers starting from 1
    for i in range(1, len(arr) + 2):
        # If the current positive integer is not in the set, return it
        if i not in elements:
            return i

# Example usage
arr = [3, 4, -1, 1]
print("Smallest positive missing number:", 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.**



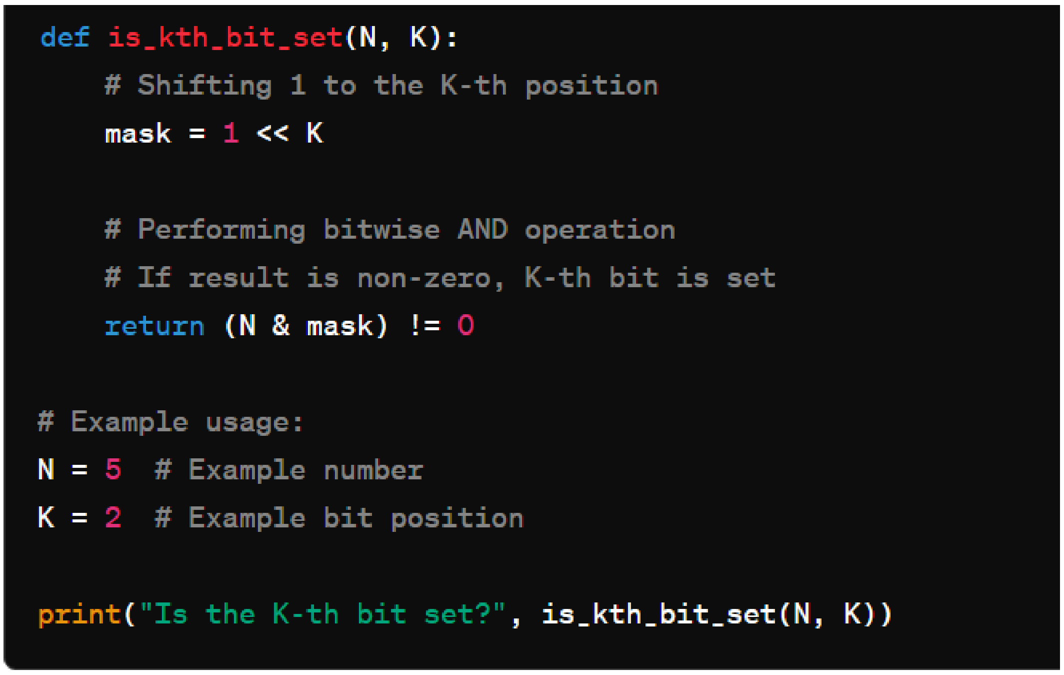
```python
def is_power_of_2(N):
    # If N is 0 or negative, it can't be a power of 2
    if N <= 0:
        return False

    # If N has only one set bit, it is a power of 2
    return (N & (N - 1)) == 0

# Example usage:
N = 16
print("Is", N, "a power of 2?", is_power_of_2(N))
```

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.**



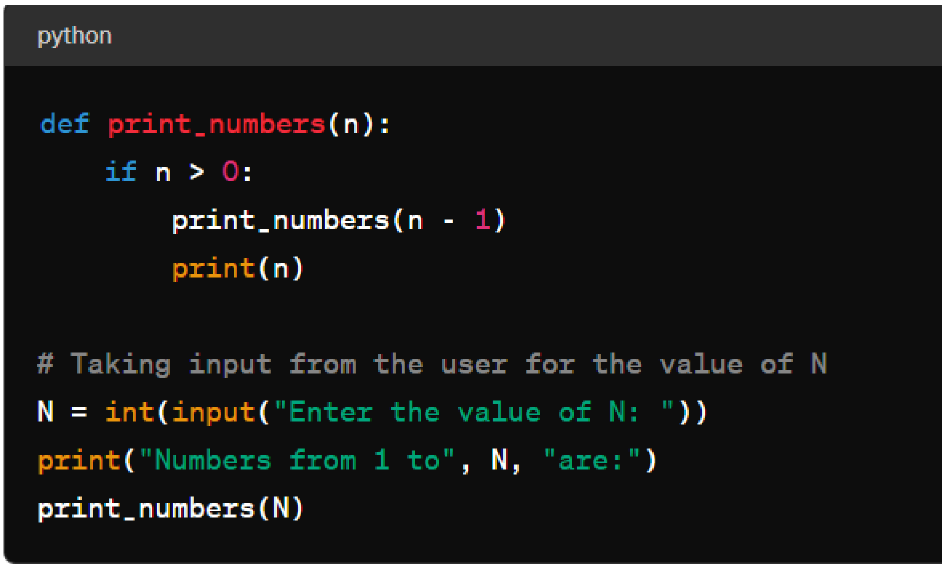
```python
def count_digits(n):
    # Base case: if n is less than 10, it means it's a single digit number
    if n < 10:
        return 1
    # Recursive case: remove the last digit from n and call count_digits recursively
    return 1 + count_digits(n // 10)

# Example usage:
n = 12345
print("Total digits in", n, ":", 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**



```python
def is_kth_bit_set(N, K):
    # Shifting 1 to the K-th position
    mask = 1 << K

    # Performing bitwise AND operation
    # If result is non-zero, K-th bit is set
    return (N & mask) != 0

# Example usage:
N = 5  # Example number
K = 2  # Example bit position

print("Is the K-th bit set?", is_kth_bit_set(N, K))
```

45 **Write a program to print 1 To N without loop**



python

```python
def print_numbers(n):
    if n > 0:
        print_numbers(n - 1)
        print(n)

# Taking input from the user for the value of N
N = int(input("Enter the value of N: "))
print("Numbers from 1 to", N, "are:")
print_numbers(N)
```