

## PRACTICE QUESTIONS FOR MTE

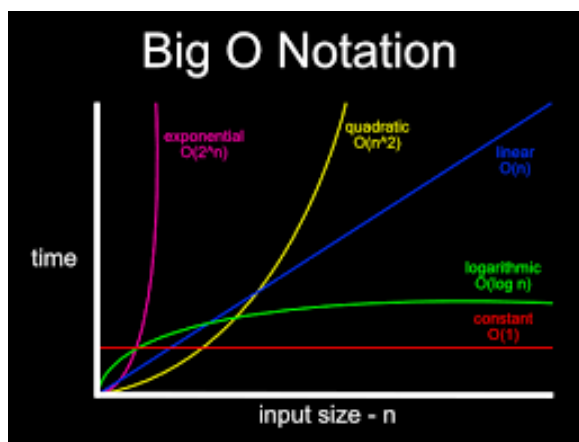
1.	<p><b>What is meant by time complexity and space complexity? Explain in detail.</b></p> <p><b>Time Complexity</b> and <b>Space Complexity</b> are two important aspects of evaluating the efficiency of an algorithm.</p> <p><b>Time Complexity</b></p> <p>Time Complexity is a measure of the amount of time an algorithm takes to run as a function of the size of the input to the program. It's usually expressed using Big O notation, which describes the upper bound of the time complexity in the worst-case scenario.</p> <p>For example, for a simple for loop iterating over <math>n</math> elements, the time complexity would be <math>O(n)</math>. This means that if the size of the input (<math>n</math>) doubles, the time taken for the algorithm also doubles.</p> <p><b>Space Complexity</b></p> <p>Space Complexity is a measure of the amount of memory an algorithm needs to run to completion. It's also expressed using Big O notation, which describes the upper bound of the space complexity in the worst-case scenario.</p> <p>For instance, consider an algorithm that creates a new list and adds <math>n</math> elements to it. The space complexity for this algorithm is <math>O(n)</math>. This means if the size of the input (<math>n</math>) doubles, the memory required by the algorithm also doubles.</p> <p>In summary, time complexity and space complexity help us to estimate the worst-case scenario in terms of time and space respectively, for an algorithm. They are crucial in determining the scalability and efficiency of an algorithm. However, there's often a trade-off between the two. For example, an algorithm might be able to run faster (lower time complexity) by using more memory (higher space complexity), and vice versa. This trade-off is often referred to as the time-space trade-off.</p>
2.	<p><b>What are asymptotic notations? Define Theta, Omega, big O, small omega, and small o.</b></p> <p>Asymptotic notations are mathematical tools used to describe the limiting behavior of functions when the argument tends towards a particular value or infinity, most often in terms of simpler functions. They are widely used in computer science to describe the time and space complexity of algorithms. Here are the definitions of the notations you asked about:</p> <ol style="list-style-type: none"><li><b>Big O Notation (<math>O</math>):</b> The Big O notation upper bounds a function to within a constant factor. It is used to describe the worst-case scenario of an algorithm's time or space complexity. For example, if we say a function <math>f(n)</math> is <math>O(g(n))</math>, it means there are some constant <math>c</math> and some value of <math>n</math> (<math>n_0</math>) after which <math>f(n)</math> is always less than or equal to <math>c \cdot g(n)</math>.</li><li><b>Omega Notation (<math>\Omega</math>):</b> The Omega notation provides an asymptotic lower bound. It is used to describe the best-case scenario of an algorithm's time or space complexity. For example, if we say a function <math>f(n)</math> is <math>\Omega(g(n))</math>, it means there are some constant <math>c</math> and some value of <math>n</math> (<math>n_0</math>) after which <math>f(n)</math> is always greater than or equal to <math>c \cdot g(n)</math>.</li><li><b>Theta Notation (<math>\Theta</math>):</b> The Theta notation bounds a function from above and below, so it defines exact asymptotic behavior. A function <math>f(n)</math> has a <math>\Theta(g(n))</math> if it is both <math>O(g(n))</math> and <math>\Omega(g(n))</math>.</li><li><b>Small o notation (<math>o</math>):</b> The small o notation is an upper bound that is not tight. If <math>f(n)</math> is <math>o(g(n))</math>, then for any constant <math>c &gt; 0</math>, there exists a constant <math>n_0</math> such that <math>0 \leq f(n) &lt; c \cdot g(n)</math> for all <math>n &gt; n_0</math>. This means <math>g(n)</math> grows strictly faster than <math>f(n)</math>.</li><li><b>Small omega notation (<math>\omega</math>):</b> The small omega notation is a lower bound that is not tight. If <math>f(n)</math> is <math>\omega(g(n))</math>, then for any constant <math>c &gt; 0</math>, there exists a constant <math>n_0</math> such that <math>0 \leq c \cdot g(n) &lt; f(n)</math> for all <math>n &gt; n_0</math>. This means <math>f(n)</math> grows strictly faster than <math>g(n)</math>.</li></ol>

## For [More Click](#)

### Advanced Algorithmic Problem Solving(R1UC601B)

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

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



4. Explain sliding window protocol.

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

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

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

There are basically two types of sliding window:

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

# For [More Click](#)

## Advanced Algorithmic Problem Solving(R1UC601B)

5.	<p><b>Explain Naive String-Matching algorithm. Discuss its time and space complexity.</b></p> <p>The Naive String-Matching Algorithm is a simple method used for pattern searching within a text. Given a text string of length <math>n</math> and a pattern of length <math>m</math>, the task is to print all occurrences of the pattern in the text. You may assume that <math>n &gt; m</math>.</p> <p>Here's how the algorithm works:</p> <ol style="list-style-type: none"> <li>1. Slide the pattern over the text one by one.</li> <li>2. For each character in the text string, check for a pattern match.</li> <li>3. If a match is found, then slide by 1 again to check for subsequent matches.</li> </ol> <p>Now, let's discuss its time and space complexity:</p> <p><b>Time Complexity:</b></p> <ul style="list-style-type: none"> <li>• <b>Best Case:</b> <math>O(n)</math> - This is when the pattern is found at the very beginning of the text (or very early on). The algorithm will perform a constant number of comparisons, typically on the order of <math>O(n)</math> comparisons, where <math>n</math> is the length of the pattern.</li> <li>• <b>Worst Case:</b> <math>O(n*m)</math> - This is when the pattern doesn't appear in the text at all or appears only at the very end. The algorithm will perform <math>O((n-m+1)*m)</math> comparisons, where <math>n</math> is the length of the text and <math>m</math> is the length of the pattern. In the worst case, for each position in the text, the algorithm may need to compare the entire pattern against the text.</li> </ul> <p><b>Space Complexity:</b></p> <ul style="list-style-type: none"> <li>• The space complexity of the Naive String-Matching Algorithm is <math>O(1)</math>. This is because it does not require any additional space that scales with input size.</li> </ul> <pre>def naive_string_matching(text, pattern):     n = len(text)     m = len(pattern)      # Slide the pattern over the text one by one     for i in range(n - m + 1):         j = 0          # For current index i, check for pattern match         while(j &lt; m):             if (text[i + j] != pattern[j]):                 break             j += 1          # If pattern matches at index i         if (j == m):             print("Pattern found at index", i)  # Driver's Code text = "AABAACAADAABAABA" pattern = "AABA" naive_string_matching(text, pattern)</pre>
6	<p><b>Explain Rabin Karp String-Matching algorithm. Discuss its time and space complexity.</b></p> <p>The Rabin-Karp Algorithm is a pattern searching algorithm that uses hashing to find any one of a set of pattern strings in a text. It uses a rolling hash to quickly filter out positions of the text that cannot match the pattern. This is why it's faster than the Naive String-Matching Algorithm for most cases.</p>

# For [More Click](#)

## Advanced Algorithmic Problem Solving(R1UC601B)

	<p>Here's how the algorithm works:</p> <ol style="list-style-type: none"> <li>1. Calculate the hash value of the pattern.</li> <li>2. Calculate the hash value of the first window of the text of the same length as the pattern.</li> <li>3. Compare the hash value of the pattern with the hash value of the current substring of text.</li> <li>4. If the hash values match, then only it starts matching individual characters.</li> <li>5. Slide the window in the text one character at a time, updating the hash value of the text window at each step.</li> </ol> <p>The hash value is calculated using a rolling hash function, which allows you to update the hash value for a new substring by efficiently removing the contribution of the old character and adding the contribution of the new character. This makes it possible to slide the pattern over the text and calculate the hash value for each substring without recalculating the entire hash from scratch.</p> <p>Now, let's discuss its time and space complexity:</p> <p><b>Time Complexity:</b></p> <ul style="list-style-type: none"> <li>• <b>Best Case:</b> <math>O(n + m)</math> - This is when the pattern is found at the very beginning of the text (or very early on). The algorithm will perform a constant number of comparisons, typically on the order of <math>O(n + m)</math> comparisons, where <math>n</math> is the length of the text and <math>m</math> is the length of the pattern.</li> <li>• <b>Worst Case:</b> <math>O(nm)</math> - This is when all characters of pattern and text are the same as the hash values of all the substrings of <math>T[]</math> match with the hash value of <math>P[]</math>. <u>The algorithm will perform <math>O((n-m+1)*m)</math> comparisons, where <math>n</math> is the length of the text and <math>m</math> is the length of the pattern<sup>1</sup>.</u></li> </ul> <p><b>Space Complexity:</b></p> <ul style="list-style-type: none"> <li>• The space complexity of the Rabin-Karp Algorithm is <math>O(1)</math>. <u>This is because it does not require any additional space that scales with input size<sup>2</sup>.</u></li> </ul> <pre> function RabinKarp(string s[1..n], string pattern[1..m])      hpattern := hash(pattern[1..m]); hs := hash(s[1..m])      for i from 1 to n-m+1          if hs = hpattern              if s[i..i+m-1] = pattern[1..m]                  return i              hs := hash(s[i+1..i+m])      return not found </pre>
7	<p><b>Explain Knuth Morris and Pratt String-Matching algorithm. Discuss its time and space complexity.</b></p> <p>It's a linear time algorithm used for pattern searching in strings.</p> <p>The KMP algorithm uses the observation that when a mismatch occurs, the pattern itself contains enough information to determine where the next match could begin. This avoids unnecessary comparisons in the future.</p>

## For [More Click](#)

### Advanced Algorithmic Problem Solving(R1UC601B)

	<p>Here's a high-level overview of how it works:</p> <ol style="list-style-type: none"> <li>1. <b>Preprocessing Step:</b> Compute a <b>prefix table (also known as failure function or pi array)</b> for the pattern. This table will be used to decide the next characters to match.</li> <li>2. <b>Matching Step:</b> Slide the pattern over the text one character at a time. When a mismatch is found, use the prefix table to skip over characters that we know will anyway match.</li> </ol> <p>The preprocessing step takes <b><math>O(m)</math></b> time, where <b><math>m</math></b> is the length of the pattern. The matching step takes <b><math>O(n)</math></b> time, where <b><math>n</math></b> is the length of the text. Therefore, the KMP algorithm has a time complexity of <b><math>O(m + n)</math></b>.</p> <p>The space complexity of the KMP algorithm is <b><math>O(m)</math></b>, as it requires storing the prefix table for the pattern.</p> <pre> function KMP(string s[1..n], string pattern[1..m])     Compute prefix table for pattern     i := 0; j := 0     while i &lt; n         if s[i] = pattern[j]             i++; j++             if j = m                 return i - j (match found)         else if j &gt; 0             j := prefix[j-1] (use prefix table to skip characters)         else             i++     return not found         </pre>
8	<p><b>What is a sliding window? Where this technique is used to solve the programming problems.</b></p> <p><b>ANS 4</b></p>
9	<p><b>What are bit manipulation operators? Explain and, or, not, ex-or bit-wise operators.</b></p> <p>Bit manipulation operators are used to perform operations at the bit level. These operators are commonly used in low-level programming such as kernel or driver development, as well as in performance-critical code. Here's a brief explanation of the bit manipulation operators you asked about:</p> <ol style="list-style-type: none"> <li>1. <b>AND (&amp;):</b> The bitwise AND operator takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.</li> </ol> <p>For example, 12 (1100 in binary) &amp; 10 (1010 in binary) will give 8 (1000 in binary).</p> <ol style="list-style-type: none"> <li>2. <b>OR ( ):</b> The bitwise OR operator takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.</li> </ol> <p>For example, 12 (1100 in binary)   10 (1010 in binary) will give 14 (1110 in binary).</p> <ol style="list-style-type: none"> <li>3. <b>NOT (~):</b> The bitwise NOT operator takes one number and inverts all bits of it.</li> </ol> <p>For example, ~12 (which is 1100 in binary) will give -13 (which is 0011 in binary in 2's complement form).</p> <ol style="list-style-type: none"> <li>4. <b>XOR (^):</b> The bitwise XOR (exclusive OR) operator takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.</li> </ol> <p>For example, 12 (1100 in binary) ^ 10 (1010 in binary) will give 6 (0110 in binary).</p>

## For More Click

### Advanced Algorithmic Problem Solving(R1UC601B)

10	<p style="color: red;">Why are the benefits for linked list over arrays.</p> <p>benefits of linked lists over arrays:</p> <ol style="list-style-type: none"> <li>1. <b>Dynamic Size:</b> Unlike arrays, linked lists do not have a fixed size. This makes them flexible for situations where the amount of data is not known in advance.</li> <li>2. <b>Ease of Insertion/Deletion:</b> Linked lists can insert or delete nodes at any point with <math>O(1)</math> time complexity (if we have a pointer to the node), unlike arrays which require shifting elements.</li> <li>3. <b>Efficient Memory Utilization:</b> Linked lists use memory more efficiently when elements are frequently added and removed. In an array, all elements need to be moved to fill or open up space, which can be costly in terms of time.</li> <li>4. <b>No Memory Wastage:</b> In linked lists, nodes can be allocated as and when required, thus reducing memory waste. In contrast, in arrays, we need to allocate memory beforehand.</li> <li>5. <b>Implementation of Stacks and Queues:</b> Linked lists are used to implement other abstract data types like stacks and queues.</li> </ol>
11	<p style="color: red;">Implement singly linked list.</p> <pre> class Node:     def __init__(self, data=None):         self.data = data         self.next = None  class LinkedList:     def __init__(self):         self.head = None      def insert(self, data):         if not self.head:             self.head = Node(data)         else:             cur = self.head             while cur.next:                 cur = cur.next             cur.next = Node(data)      def display(self):         cur = self.head         while cur:             print(cur.data, end=' ')             cur = cur.next         print()  # Usage ll = LinkedList() ll.insert(1) ll.insert(2) ll.insert(3) ll.display() # Outputs: 1 2 3 </pre>
11	<p style="color: red;">Implement stack with singly linked list.</p> <pre> class Node:     def __init__(self, data=None):         self.data = data         self.next = None  class Stack:     def __init__(self):         self.top = None      def push(self, data):         new_node = Node(data)         new_node.next = self.top </pre>

## For [More Click](#)

### Advanced Algorithmic Problem Solving(R1UC601B)

	<pre>self.top = new_node  def pop(self):     if self.top is None:         return None     else:         popped_data = self.top.data         self.top = self.top.next         return popped_data  def peek(self):     return self.top.data if self.top else None  # Usage s = Stack() s.push(1) s.push(2) s.push(3) print(s.pop()) # Outputs: 3 print(s.peek()) # Outputs: 2</pre>
12	<p><b>Implement queue with singly linked list.</b></p> <pre>class Node:     def __init__(self, data=None):         self.data = data         self.next = None  class Queue:     def __init__(self):         self.front = self.rear = None      def is_empty(self):         return self.front is None      def enqueue(self, data):         new_node = Node(data)         if self.rear is None:             self.front = self.rear = new_node             return         self.rear.next = new_node         self.rear = new_node      def dequeue(self):         if self.is_empty():             return None         temp = self.front         self.front = temp.next         if self.front is None:             self.rear = None         return temp.data  # Usage q = Queue() q.enqueue(1) q.enqueue(2) q.enqueue(3) print(q.dequeue()) # Outputs: 1</pre>
12	<p><b>Implement doubly linked list.</b></p> <pre>class Node:     def __init__(self, data=None):         self.data = data         self.next = None</pre>

## For [More Click](#)

### Advanced Algorithmic Problem Solving(R1UC601B)

	<pre> self.prev = None  class DoublyLinkedList:     def __init__(self):         self.head = None      def append(self, data):         if self.head is None:             self.head = Node(data)         else:             new_node = Node(data)             cur = self.head             while cur.next:                 cur = cur.next             cur.next = new_node             new_node.prev = cur      def prepend(self, data):         if self.head is None:             self.head = Node(data)         else:             new_node = Node(data)             new_node.next = self.head             self.head.prev = new_node             self.head = new_node      def print_list(self):         cur = self.head         while cur:             print(cur.data)             cur = cur.next </pre>
13	<p><b>Implement circular linked list.</b></p> <pre> class Node:     def __init__(self, data=None):         self.data = data         self.next = None  class CircularLinkedList:     def __init__(self):         self.head = None      def append(self, data):         if not self.head:             self.head = Node(data)             self.head.next = self.head         else:             new_node = Node(data)             cur = self.head             while cur.next != self.head:                 cur = cur.next             cur.next = new_node             new_node.next = self.head      def print_list(self):         cur = self.head         while True:             print(cur.data)             cur = cur.next             if cur == self.head:                 break </pre>
14	<p><b>Implement circular queue with linked list.</b></p>



# For [More Click](#)

## Advanced Algorithmic Problem Solving(R1UC601B)

	<pre> class Node:     def __init__(self, data):         self.data = data         self.next = None  class CircularQueue:     def __init__(self):         self.front = self.rear = None      def enqueue(self, data):         temp = Node(data)          if self.rear == None:             self.front = self.rear = temp             self.rear.next = self.front         else:             self.rear.next = temp             self.rear = temp             self.rear.next = self.front      def dequeue(self):         if self.front == None:             print("Circular Queue is empty")             return          temp = self.front         self.front = temp.next         self.rear.next = self.front          if self.front == self.rear:             self.front = self.rear = None      def display(self):         temp = self.front         while temp:             print(temp.data, end=" ")             temp = temp.next             if temp == self.front:                 break </pre>
15	<p><b>What is recursion? What is tail recursion?</b></p> <p>Recursion is a process in which a function calls itself as a subroutine. This allows the function to be broken down into smaller, more manageable problems that are easier to solve<sup>1</sup>. It's a method of solving a computational problem where the solution depends on solutions to smaller instances of the same problem<sup>2</sup>. Recursion has some important properties<sup>1</sup>:</p> <p>The ability to solve a problem by breaking it down into smaller sub-problems.  A recursive function must have a base case or stopping criteria to avoid infinite recursion.  Recursive functions may be less efficient than iterative solutions in terms of memory and performance.</p> <p>Tail recursion is a special case of recursion where the recursive call is the last operation in the recursive function<sup>3</sup>. What this means is that all calculations are performed first, and the execution of the recursive call is the final operation<sup>3</sup>. This is important because it allows certain compilers to optimize the recursion, using a loop construct instead of a function call for the recursive step<sup>3</sup>. This can result in significant savings in both processing time and stack space<sup>3</sup>.</p> <p>Here's an example of a tail-recursive function in Python:</p>

# For [More Click](#)

## Advanced Algorithmic Problem Solving(R1UC601B)

	<p>Python</p> <pre>def factorial(n, acc=1):     if n == 0:         return acc     else:         return factorial(n-1, n*acc)</pre> <p>In this example, factorial(n-1, n*acc) is the last operation in the recursive function, making it tail recursive. The acc parameter accumulates the result of the factorial calculation, which is returned when n reaches 0. This is the base case of the recursion. The advantage of this tail-recursive version is that it uses constant stack space and is more efficient.</p>
16	<p><b>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.</b></p> <p>The <b>Tower of Hanoi</b> problem is a classic problem in the field of computer science and mathematics. It consists of three rods and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.</p> <p>The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:</p> <ol style="list-style-type: none"> <li>1. Only one disk can be moved at a time.</li> <li>2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.</li> <li>3. No disk may be placed on top of a smaller disk.</li> </ol> <pre>def TowerOfHanoi(n , source, destination, auxiliary):     if n==1:         print("Move disk 1 from source",source,"to destination",destination)         return     TowerOfHanoi(n-1, source, auxiliary, destination)     print("Move disk",n,"from source",source,"to destination",destination)     TowerOfHanoi(n-1, auxiliary, destination, source)  # Driver code n = 4 TowerOfHanoi(n,'A','B','C') # A, C, B are the name of rods</pre> <p>The <b>time complexity</b> of the Tower of Hanoi problem is <math>O(2^n)</math>, where n is the number of disks. This is because with each increase in the number of disks, the number of steps required to solve the problem doubles.</p> <p>The <b>space complexity</b> of the Tower of Hanoi problem is <math>O(n)</math>, where n is the number of disks. This is due to the recursive nature of the problem, which results in n recursive calls, and thus n items in the call stack. Each recursive call requires a constant amount of space.</p>
17	<p><b>What is backtracking in algorithms? What kind of problems are solved with this technique?</b></p> <p>Backtracking is a general algorithmic technique that involves finding a solution incrementally by trying different options and undoing them if they lead to a dead end. It is commonly used in situations where you need to explore multiple possibilities to solve a problem, like searching for a path in a maze or solving puzzles like Sudoku.</p> <p>A backtracking algorithm works by recursively exploring all possible solutions to a problem. It</p>

## For [More Click](#)

### Advanced Algorithmic Problem Solving(R1UC601B)

starts by choosing an initial solution, and then it explores all possible extensions of that solution. If an extension leads to a solution, the algorithm returns that solution. If an extension does not lead to a solution, the algorithm backtracks to the previous solution and tries a different extension<sup>1</sup>.

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

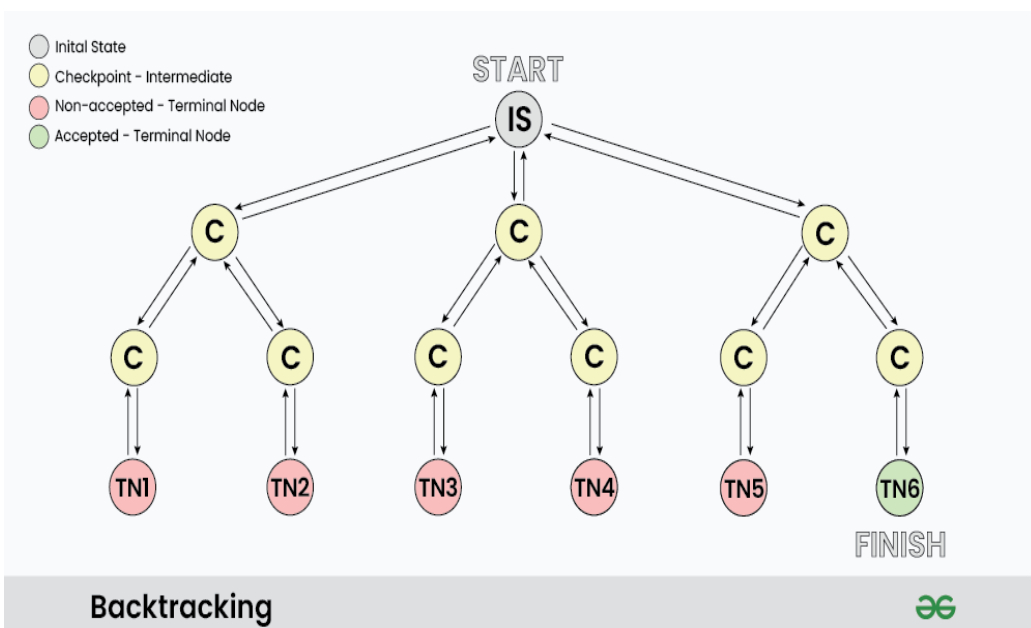
Puzzles: Such as the N-Queens problem, Sudoku, and the Knight's tour problem<sup>12</sup>.

Path Finding: Finding the shortest path through a maze or finding all paths from source to destination in a matrix<sup>12</sup>.

Combinatorial Problems: Generating all possible combinations of a set of items<sup>3</sup>.

Constraint Satisfaction Problems: Finding a specific solution that satisfies a set of constraints<sup>3</sup>.

Remember, while backtracking can solve these problems, it may not always be the most efficient method, especially for problems with large input sizes or many potential solutions. Other techniques like dynamic programming might be more suitable in such cases<sup>4</sup>.



18

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

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

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

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

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

    return True

def solveNQUtil(board, col, N):
```

## For [More Click](#)

### Advanced Algorithmic Problem Solving(R1UC601B)

```
# base case: If all queens are placed
if col >= N:
    return True

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

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

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

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

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

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

    printSolution(board)
    return True

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

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

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

## For [More Click](#)

### Advanced Algorithmic Problem Solving(R1UC601B)

19	<p><b>What is subset sum problem? Write a recursive function to solve the subset sum problem?</b></p> <p>The <b>Subset Sum Problem</b> is a classic computer science problem that falls under the category of <b>NP-Complete problems</b>. It is a decision problem that can be stated as follows:</p> <p>Given a set of integers and an integer <b>S</b>, does any subset of the given set add up to <b>S</b>?</p> <pre>def is_subset_sum(set, n, sum):     # Base Cases     if sum == 0:         return True     if n == 0 and sum != 0:         return False      # If last element is greater than sum, then ignore it     if set[n-1] &gt; sum:         return is_subset_sum(set, n-1, sum)      # Else, check if sum can be obtained by any of the following:     # (a) including the last element     # (b) excluding the last element     return is_subset_sum(set, n-1, sum) or is_subset_sum(set, n-1, sum-set[n-1])  # Test the function set = [3, 34, 4, 12, 5, 2] sum = 9 n = len(set) if (is_subset_sum(set, n, sum) == True):     print("Found a subset with given sum") else:     print("No subset with given sum")</pre>
20	<p><b>Implement a function that uses the sliding window technique to find the maximum sum of any contiguous subarray of size K.</b></p> <pre>def max_sum_subarray_size_k(arr, k):     window_sum = 0     max_sum = 0     window_start = 0      for window_end in range(len(arr)):         window_sum += arr[window_end] # add the next element          # slide the window, we don't need to slide if we've not hit the required window size of 'k'         if window_end &gt;= k-1:             max_sum = max(max_sum, window_sum)             window_sum -= arr[window_start] # subtract the element going out             window_start += 1 # slide the window ahead      return max_sum  # Test the function arr = [2, 3, 4, 1, 5] k = 3 print("Maximum sum of a subarray of size K: ", max_sum_subarray_size_k(arr, k))</pre>

## For [More Click](#)

### Advanced Algorithmic Problem Solving(R1UC601B)

21	<p>Write a recursive function to generate all possible subsets of a given set.</p> <pre>def generate_subsets(s, current_subset=[], index=0):     if index == len(s):         print(current_subset)     else:         # Two possibilities for each element, it's either in the subset or not         # First, we select the element to be in the subset         generate_subsets(s, current_subset + [s[index]], index + 1)         # Then, we do not select the element to be in the subset         generate_subsets(s, current_subset, index + 1)  # Test the function s = [1, 2, 3] generate_subsets(s)</pre>
22	<p>Write a program to find the first occurrence of repeating character in a given string.</p> <pre>def first_repeating_char(s):     char_count = {}     for char in s:         if char in char_count:             return char         else:             char_count[char] = 1     return None  # Test the function s = "interviewquery" print("The first repeating character is: ", first_repeating_char(s))</pre>
23	<p>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.</p> <pre>def print_leaders(arr):     max_from_right = arr[-1] # The rightmost element is always a leader     print(max_from_right, end=' ')     for i in range(len(arr)-2, -1, -1):         if max_from_right &lt;= arr[i]: # If this element is greater than the max from right             print(arr[i], end=' ') # This element is a leader             max_from_right = arr[i] # Update the max from right  # Test the function arr = [16, 17, 4, 3, 5, 2] print("Leaders in the array are: ", end="") print_leaders(arr)</pre>
24	<p>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.</p> <pre>def find_majority_element(arr):     majority_element = 0     count = 0      # Boyer-Moore Voting Algorithm     for i in range(len(arr)):         if count == 0:             majority_element = arr[i]         if arr[i] == majority_element:             count += 1</pre>

## For [More Click](#)

### Advanced Algorithmic Problem Solving(R1UC601B)

	<pre>else:     count -= 1  # Verify if the majority_element appears more than n/2 times count = arr.count(majority_element) if count &gt; len(arr) // 2:     return majority_element else:     return "No Majority Element"  # Test the function arr = [2, 2, 1, 1, 1, 2, 2] print(find_majority_element(arr))</pre>
25	<p>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.</p> <pre>from queue import Queue from collections import deque  def reverse_first_k(queue, k):     if queue.empty() or k &gt; queue.qsize():         return "Invalid input"      stack = deque()     for _ in range(k):         stack.append(queue.get())     while stack:         queue.put(stack.pop())     for _ in range(queue.qsize() - k):         queue.put(queue.get())      return queue  # Test the function queue = Queue() for i in range(1, 11):     queue.put(i)  reverse_first_k(queue, 5)  while not queue.empty():     print(queue.get(), end=' ')</pre>
26	<p>Write a program to implement a stack using queues.</p> <pre>class Stack:     def __init__(self):         self.q1 = []         self.q2 = []      def push(self, x):         self.q1.append(x)      def pop(self):         if not self.q1:             return "Stack is empty"         while len(self.q1) &gt; 1:             self.q2.append(self.q1.pop(0))         item = self.q1.pop()         self.q1, self.q2 = self.q2, self.q1         return item</pre>

## For [More Click](#)

### Advanced Algorithmic Problem Solving(R1UC601B)

	<pre># Test the Stack stack = Stack() stack.push(1) stack.push(2) stack.push(3) print(stack.pop()) # prints 3 print(stack.pop()) # prints 2</pre>
27	<p>Write a program to implement queue using stacks.</p> <pre>class Queue:     def __init__(self):         self.s1 = []         self.s2 = []      def enqueue(self, x):         self.s1.append(x)      def dequeue(self):         if not self.s1 and not self.s2:             return "Queue is empty"         if not self.s2:             while self.s1:                 self.s2.append(self.s1.pop())         return self.s2.pop()  # Test the Queue queue = Queue() queue.enqueue(1) queue.enqueue(2) queue.enqueue(3) print(queue.dequeue()) # prints 1 print(queue.dequeue()) # prints 2</pre>
28	<p>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.</p> <pre>def is_isogram(string):     # Convert the string to lowercase for uniformity     string = string.lower()      # Create a set from the string. In a set, duplicate elements are not allowed.     # So, if the length of the set is equal to the length of the string, it means all characters were unique.     return len(string) == len(set(string))  # Test the function print(is_isogram("subdermatoglyphic")) # This should return True print(is_isogram("hello")) # This should return False</pre>
29	<p>Given a sorted <u>array</u>, arr[] consisting of N integers, write a program to find the <u>frequencies of each array element</u>.</p> <pre>def count_frequencies(arr):     # Initialize the count and the element     count = 1     element = arr[0]      # Iterate over the array     for i in range(1, len(arr)):         # If the current element is the same as the previous one, increment the count         if arr[i] == element:             count += 1         else:             # Print the element and its frequency             print(f"Element {element} occurs {count} times")</pre>



## For [More Click](#)

### Advanced Algorithmic Problem Solving(R1UC601B)

	<pre># Update the element and reset the count element = arr[i] count = 1  # Print the last element and its frequency print(f"Element {element} occurs {count} times")  # Test the function arr = [1, 1, 2, 2, 2, 3, 4, 4, 4, 4] count_frequencies(arr)</pre>
30	<p><b>Write a program to delete middle element from stack.</b></p> <pre>def delete_middle(stack, n, curr=0):     # Base case: If stack is empty or all items are traversed     if not stack or curr == n:         return      # Remove current item     temp = stack.pop()      # Remove remaining items     delete_middle(stack, n, curr+1)      # Put all items back except middle     if curr != n//2:         stack.append(temp)  # Test the function stack = [1, 2, 3, 4, 5] delete_middle(stack, len(stack)) print(stack) # This should print [1, 2, 4, 5]</pre>
31	<p><b>Write a program to remove consecutive duplicates from string.</b></p> <pre>def remove_consecutive_duplicates(s):     # Initialize the result     result = ""      # Iterate over the string     for i in range(len(s)):         # If the current character is not the same as the previous one, add it to the result         if i == 0 or s[i] != s[i-1]:             result += s[i]      return result  # Test the function print(remove_consecutive_duplicates("aaabbbccdaaa")) # This should print "abcda"</pre>
33	<p><b>Write a program to display next greater element of all element given in array.</b></p> <pre>def print_next_greater(arr):     stack = []     for i in range(len(arr)):         while stack and stack[-1] &lt; arr[i]:             print(f"Next greater element for {stack.pop()} is {arr[i]}")         stack.append(arr[i])     while stack:         print(f"Next greater element for {stack.pop()} is -1")  # Test the function arr = [4, 5, 2, 25] print_next_greater(arr)</pre>

## For [More Click](#)

### Advanced Algorithmic Problem Solving(R1UC601B)

34

Write a program to evaluate a postfix expression.

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

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

## For [More Click](#)

Advanced Algorithmic Problem Solving(R1UC601B)

35	<p>Write a program to get MIN at pop from stack.</p> <pre>class MinStack:     def __init__(self):         self.stack = []         self.min_stack = []      def push(self, x):         self.stack.append(x)         if not self.min_stack or x &lt;= self.min_stack[-1]:             self.min_stack.append(x)      def pop(self):         if self.stack:             top = self.stack.pop()             if self.min_stack and self.min_stack[-1] == top:                 self.min_stack.pop()             return top      def get_min(self):         return self.min_stack[-1] if self.min_stack else None  # Test the MinStack min_stack = MinStack() min_stack.push(3) min_stack.push(5) print(min_stack.get_min()) # prints 3 print(min_stack.pop()) # prints 5 print(min_stack.get_min()) # prints 3</pre>
36	<p>Write a program to swap <math>k^{\text{th}}</math> node from ends in given single linked list.</p> <pre>class Node:     def __init__(self, data):         self.data = data         self.next = None  class LinkedList:     def __init__(self):         self.head = None      def push(self, data):         new_node = Node(data)         new_node.next = self.head         self.head = new_node      def print_list(self):         node = self.head         while node:             print(node.data, end=" ")             node = node.next         print()      def swap_kth_node(self, k):         n = 0         node = self.head         while node:             n += 1             node = node.next          if n &lt; k:             return          x = self.head         x_prev = None</pre>

## For [More Click](#)

### Advanced Algorithmic Problem Solving(R1UC601B)

	<pre>for i in range(k - 1):     x_prev = x     x = x.next  y = self.head y_prev = None for i in range(n - k):     y_prev = y     y = y.next  if x_prev:     x_prev.next = y else:     self.head = y  if y_prev:     y_prev.next = x else:     self.head = x  temp = x.next x.next = y.next y.next = temp  # Test the LinkedList l1 = LinkedList() for i in range(8, 0, -1):     l1.push(i) print("Original linked list:") l1.print_list() l1.swap_kth_node(3) print("Linked list after swapping 3rd node from both ends:") l1.print_list()</pre>
37	<p>Write a program to detect loop in linked list</p> <pre>class Node:     def __init__(self, data):         self.data = data         self.next = None  class LinkedList:     def __init__(self):         self.head = None      def push(self, data):         new_node = Node(data)         new_node.next = self.head         self.head = new_node      def detect_loop(self):         slow_p = self.head         fast_p = self.head         while slow_p and fast_p and fast_p.next:             slow_p = slow_p.next             fast_p = fast_p.next.next             if slow_p == fast_p:                 return True         return False  # Test the LinkedList l1 = LinkedList() for i in range(5, 0, -1):</pre>

## For [More Click](#)

### Advanced Algorithmic Problem Solving(R1UC601B)

	<pre>l1.push(i) # Create a loop for testing l1.head.next.next.next.next.next = l1.head.next.next print("Loop detected:" if l1.detect_loop() else "No loop detected")</pre>
38	<p><b>Write a program to find Intersection point in Y shaped Linked list.</b></p> <pre>class Node:     def __init__(self, data):         self.data = data         self.next = None  def getIntersectionNode(head1, head2):     curr1 = head1     curr2 = head2      # Traverse through both lists. If one list ends, start at the beginning of the other list.     # If there is an intersection, the pointers should meet at the intersection point after 2     traversals.     while curr1 != curr2:         curr1 = curr1.next if curr1 else head2         curr2 = curr2.next if curr2 else head1      return curr1.data # Return the intersection node  # Test the function with an example # Create a Y shaped linked list head1 = Node(3) head1.next = Node(6) head1.next.next = Node(9) head1.next.next.next = Node(15) head1.next.next.next.next = Node(30)  head2 = Node(10) head2.next = head1.next.next.next # Intersection point  print("Intersection point is", getIntersectionNode(head1, head2))</pre>
39	<p><b>Write a program to merge two sorted linked list.</b></p> <pre>class Node:     def __init__(self, data):         self.data = data         self.next = None  def mergeLists(head1, head2):     # A dummy node to store the result     dummyNode = Node(0)      # last stores the last node     last = dummyNode     while True:          # If either list runs out, use the other list         if head1 is None:             last.next = head2             break         if head2 is None:             last.next = head1             break          # Compare the data of the lists and whichever is smaller is appended to the last's         next and the head is changed</pre>

## For [More Click](#)

### Advanced Algorithmic Problem Solving(R1UC601B)

	<pre>if head1.data &lt;= head2.data:     last.next = head1     head1 = head1.next else:     last.next = head2     head2 = head2.next  # Update the last for next insertion last = last.next  # Returns the head of the merged list return dummyNode.next  # Test the function with an example # Create two sorted linked lists head1 = Node(1) head1.next = Node(2) head1.next.next = Node(4)  head2 = Node(1) head2.next = Node(3) head2.next.next = Node(4)  merged_head = mergeLists(head1, head2)  # Print the merged list while merged_head is not None:     print(merged_head.data, end=" ")     merged_head = merged_head.next</pre>
40	<p><b>Write a program to find max and second max of array.</b></p> <pre>def find_max_and_second_max(arr):     if len(arr) &lt; 2:         return "Invalid input. Array should have at least two elements."      max1 = max2 = float('-inf')     for num in arr:         if num &gt; max1:             max2 = max1             max1 = num         elif num &gt; max2 and num != max1:             max2 = num      if max2 == float('-inf'):         return "No second max value, all elements are the same."     else:         return max1, max2  # Test the function with an example arr = [2, 3, 6, 6, 5] print("Max and second max of array:", find_max_and_second_max(arr))</pre>
41	<p><b>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.</b></p> <pre>def find_smallest_missing_positive(arr):     if not arr:         return 1      arr = set(arr)     smallest_positive = 1      while smallest_positive in arr:</pre>

## For [More Click](#)

### Advanced Algorithmic Problem Solving(R1UC601B)

	<pre> smallest_positive += 1  return smallest_positive  # Test the function with an example arr = [0, 10, 2, -10, -20, 1, 3] print("Smallest positive missing number:", find_smallest_missing_positive(arr)) </pre>
42	<p><b>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.</b></p> <pre> def is_power_of_two(n):     if n &lt;= 0:         return False     else:         return (n &amp; (n - 1)) == 0  # Test the function with an example n = 16 print("Is", n, "a power of 2?", is_power_of_two(n)) </pre> <ol style="list-style-type: none"> <li><b>If n is less than or equal to 0:</b> The function returns <code>False</code>. This is because powers of 2 are always positive and non-zero.</li> <li><b>If n is greater than 0:</b> The function checks if <code>n AND n-1</code> is equal to 0. This is a bitwise operation. <ul style="list-style-type: none"> <li>The bitwise AND operation (<code>&amp;</code>) compares each bit of the first operand (<code>n</code>) to the corresponding bit of the second operand (<code>n-1</code>). If both bits are 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.</li> <li>For numbers that are powers of 2, their binary representation has a single '1' bit and the rest are '0'. For example, 2 is <code>10</code> in binary and 4 is <code>100</code> in binary.</li> <li>When you subtract 1 from these numbers, you get a binary number with the '1' bit turned to '0' and all bits to the right of it turned to '1'. For example, <code>2-1</code> is 1 which is <code>01</code> in binary and <code>4-1</code> is 3 which is <code>011</code> in binary.</li> <li>So, when you perform a bitwise AND operation between a power of 2 and one less than it, you get 0 because there are no positions where both numbers have a '1' bit.</li> <li>Therefore, <code>(n &amp; (n - 1)) == 0</code> is a quick way to check if a number is a power of 2.</li> </ul> </li> </ol>
43	<p><b>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.</b></p> <pre> def count_digits(n):     if n == 0:         return 0     return 1 + count_digits(n // 10)  # Test the function with an example n = 12345 print("The number of digits in", n, "is", count_digits(n)) </pre>
44	<p><b>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 <b>true</b> or <b>false</b>.</b></p>

## For [More Click](#)

### Advanced Algorithmic Problem Solving(R1UC601B)

	<pre>def is_kth_bit_set(n, k):     if n &amp; (1 &lt;&lt; k):         return True     else:         return False  # Test the function with an example n = 5 # binary: 101 k = 2 print("Is the", k, "th bit of", n, "set?", is_kth_bit_set(n, k))</pre>
45	<p><b>Write a program to print 1 To N without loop</b></p> <pre>def print_numbers(n):     if n &gt; 0:         print_numbers(n - 1)         print(n)  # Test the function with an example N = 10 print("Printing numbers from 1 to", N) print_numbers(N)</pre>