**PABNA UNIVERSITY OF SCIENCE AND TECHNOLOGY**

A logo with text and symbols

AI-generated content may be incorrect.

**Faculty of Engineering & Technology**

**Department of Information and Communication Engineering**

LAB REPORT

**Course name: Data Structure and Algorithm Sessional**

**Course Code: ICE-2202**

Submitted By: Submitted To:

**MD. Junaid Md. Anwar Hossain**

Roll: 220618 Professor

Session: 2021-22 Department of ICE

2nd Year 2nd Semester Pabna University of Science and

Department of ICE, PUST Technology, Pabna

Date of Submission: 28-02-2025

|  |  |
| --- | --- |
| **Sl.** | **Problem Statement** |
| **1.** | |  | | --- | | Write a program to sort a linear array using the bubble sort algorithm. | |
| **2.** | |  | | --- | | Write a program to find an element using a linear search algorithm. | |
| **3.** | |  | | --- | | Write a program to sort a linear array using the merge sort algorithm. | |
| **4.** | |  | | --- | | Write a program to find an element using the binary search algorithm. | |
| **5.** | |  | | --- | | Write a program to find a given pattern from text using the pattern matching algorithm. | |
| **6.** | |  | | --- | | Write a program to implement a queue data structure along with its typical operations. | |
| **7.** | |  | | --- | | Write a program to solve **n** queen's problem using backtracking. | |
| **8.** | |  | | --- | | Consider a set **S = {5,10,12,13,15,18}** and **d = 30**. Write a program to solve the sum of subset problem. | |
| **9.** | |  | | --- | | Write a program to solve the following **0/1 Knapsack** using dynamic programming approach **profits P = (15,25,13,23), weight W = (2,6,12,9), Knapsack C = 20**, and the number of items **n=4**. | |
| **10** | |  | | --- | | Write a program to solve the **Tower of Hanoi** problem for the **N** disk. | |

**Index**

**Problem 1: Sorting a Linear Array using Bubble Sort Algorithm**

**Problem Illustration and Logic**

Bubble Sort is a simple comparison-based sorting algorithm that works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order. The algorithm gets its name because smaller elements "bubble" to the top of the list (beginning of the array) with each iteration.

The logic behind bubble sort is:

1. Compare each pair of adjacent elements in the array
2. If they are in the wrong order (the left element is greater than the right element), swap them
3. Continue this process for the entire array
4. Repeat the process for n-1 passes, where n is the length of the array
5. After each pass, the largest unsorted element "bubbles up" to its correct position

The algorithm becomes more efficient with a small optimization: if no swaps occur during a pass, the array is already sorted and we can terminate early.

**Step-by-step Algorithm**

1. Start with the first element (index 0)
2. Compare the current element with the next element
3. If the current element is greater than the next element, swap them
4. Move to the next element and repeat steps 2-3 until the end of the array
5. After one complete pass, the largest element will be at the end of the array
6. Repeat steps 1-5 for all elements except the last one
7. Keep track of whether any swap occurred in a pass - if no swaps occurred, the array is sorted
8. Continue this process until the array is sorted

A diagram of a diagram

AI-generated content may be incorrect.

**Source Code**

|  |
| --- |
| def bubble\_sort(arr):      n = len(arr)      for i in range(n):          print("Pass :", i+1)          for j in range(0, n-i-1):              if(arr[j] > arr[j+1]):                  arr[j], arr[j+1] = arr[j+1], arr[j]              print(arr)      return arr  array = [64, 34, 25, 12, 22, 11, 90]  print("Sorted Array:", bubble\_sort(array)) |

**Sample Input and Output**

**Input:**

|  |
| --- |
| Original array:  64 34 25 12 22 11 90 |

**Output:**

|  |
| --- |
| Sorted array:  1 12 22 25 34 64 90 |

**Problem 2: Finding an Element using Linear Search Algorithm**

**Problem Illustration and Logic**

Linear Search is a simple searching algorithm that sequentially checks each element in a list until it finds the target element or reaches the end of the list. It's straightforward but can be inefficient for large datasets compared to other search algorithms like binary search.

The logic behind linear search is:

1. Start from the leftmost element of the array
2. Compare each element with the target value
3. If the element matches the target value, return its index
4. If the element doesn't match, move to the next element
5. If no match is found after checking all elements, return -1 to indicate the element is not present

Linear search has a time complexity of O(n) in the worst case, where n is the number of elements in the array.

**Step-by-step Algorithm**

1. Create a function that takes an array and a target value as input
2. Start from the first element (index 0)
3. Compare the current element with the target value
4. If they match, return the current index
5. If they don't match, move to the next element
6. Repeat steps 3-5 until either the element is found or the end of the array is reached
7. If the element is not found after checking all elements, return -1

A diagram of a line with numbers and arrows

AI-generated content may be incorrect.

**Source Code**

|  |
| --- |
| def linear\_search(arr, target):      for i in range(len(arr)):          if arr[i] == target:              return i      return -1  array = [15,9,35,10,1,22,7,57,17,2]  target = 22  print("Element found at index:", linear\_search(array, target)) |

**Sample Input and Output**

**Input:**

|  |
| --- |
| 15,9,35,10,1,22,7,57,17,2 |

**Output:**

|  |
| --- |
| Element found at index: 5 |

**Problem 3: Sorting a Linear Array using Merge Sort Algorithm**

**Problem Illustration and Logic**

Merge Sort is an efficient, stable, comparison-based, divide and conquer sorting algorithm. Unlike simpler algorithms like bubble sort, merge sort has a consistent O(n log n) time complexity, making it more efficient for larger datasets.

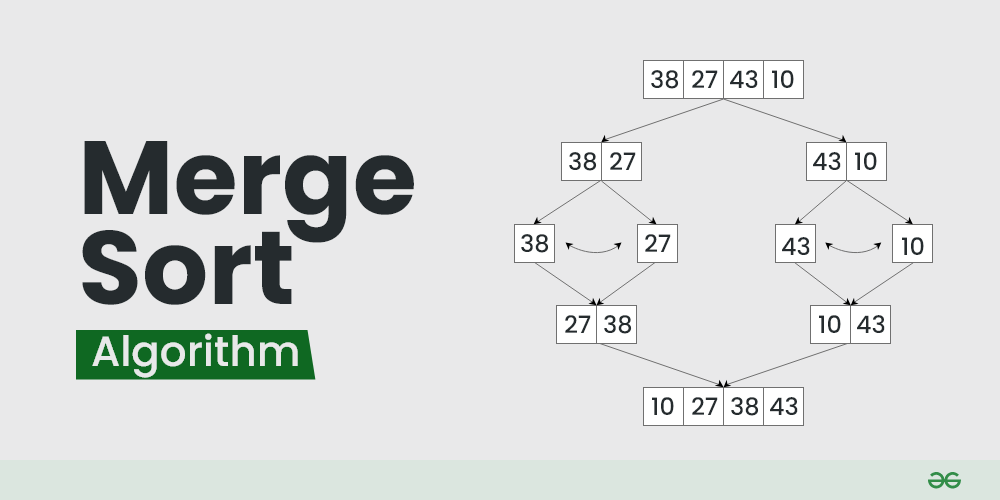
The logic behind merge sort is:

1. Divide the unsorted array into n subarrays, each containing one element (an array of one element is considered sorted)
2. Repeatedly merge subarrays to produce new sorted subarrays until only one subarray remains
3. The final single subarray is the sorted array

The key operation is the merging of two sorted subarrays into a single sorted array. This is done by comparing the smallest elements of both subarrays and placing the smaller one into the result array.

**Step-by-step Algorithm**

1. If the array has only one element or is empty, it is already sorted
2. Divide the array into two halves:
   * Find the middle point of the array
   * Split the array from the first element to the middle element
   * Split the array from the middle+1 element to the last element
3. Recursively sort both halves
4. Merge the sorted halves:
   * Create temporary arrays for both halves
   * Compare elements from both halves and place them in the correct order in the original array
   * Copy any remaining elements from either half



**Source Code**

|  |
| --- |
| **#include<bits/stdc++.h>**  **using namespace std;**  **void merge(vector<int> &arr, int st, int mid,int end){**  **int i=st, j = mid+1;**  **vector<int> tmp;**  **while(i<=mid && j<=end){**  **if(arr[i] < arr[j]){**  **tmp.push\_back(arr[i]);**  **i++;**  **}else{**  **tmp.push\_back(arr[j]);**  **j++;**  **}**  **}**  **while(i<=mid){**  **tmp.push\_back(arr[i]);**  **i++;**  **}**  **while(j<=end){**  **tmp.push\_back(arr[j]);**  **j++;**  **}**  **for(int indx = 0; indx<tmp.size(); indx++){**  **arr[indx+st] = tmp[indx];**  **}**  **}**  **void mergeSort(vector<int> &arr, int st, int end){**  **if(st < end){**  **int mid = st + (end-st)/2;**  **mergeSort(arr, st, mid);**  **mergeSort(arr, mid+1, end);**  **merge(arr, st, mid, end);**  **}**  **}**  **int main(){**  **vector<int> arr = {12, 13, 43, 1, 2, 56};**  **mergeSort(arr, 0, arr.size()-1);**  **for(int val: arr){**  **cout << val <<" ";**  **}**  **return 0;**  **}** |

**Sample Input and Output**

**Input:**

|  |
| --- |
| **12, 13, 43, 1, 2, 56** |

**Output:**

|  |
| --- |
| **1 2 12 13 43 56** |

**Problem 4: Finding an Element using Binary Search Algorithm**

**Problem Illustration and Logic**

Binary Search is an efficient searching algorithm that works on sorted arrays. Unlike linear search which checks each element sequentially, binary search divides the search space in half with each comparison, resulting in a logarithmic time complexity of O(log n).

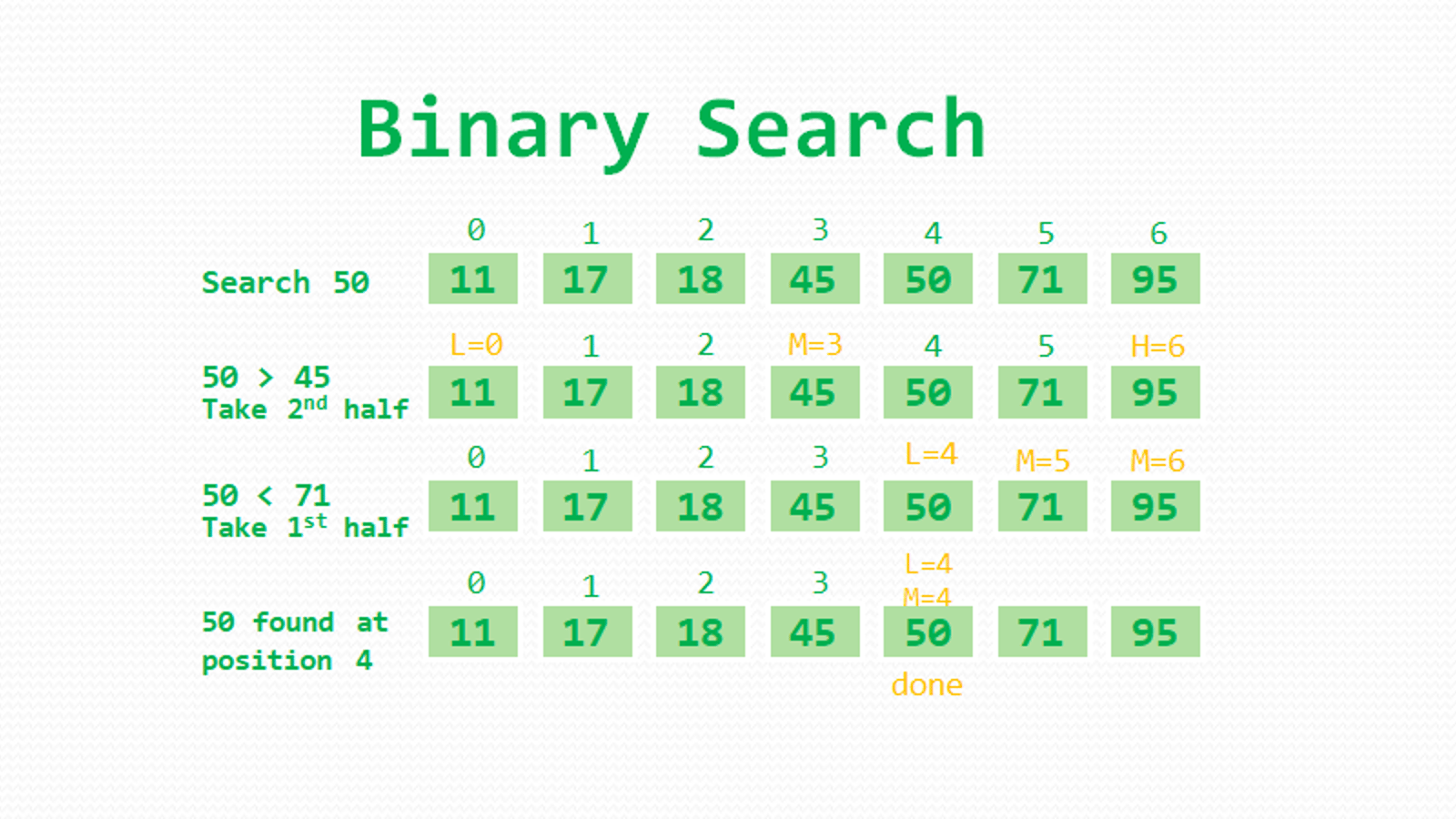
The logic behind binary search is:

1. Start with the entire sorted array
2. Find the middle element of the current search range
3. If the target equals the middle element, the search is complete
4. If the target is less than the middle element, search the left half
5. If the target is greater than the middle element, search the right half
6. Repeat steps 2-5 until the element is found or the search space is empty

Binary search requires the array to be sorted beforehand, which is a prerequisite for the algorithm to work correctly.

**Step-by-step Algorithm**

1. Initialize two pointers: left pointing to the first element and right pointing to the last element
2. While left is less than or equal to right:
   * Calculate the middle index as mid = left + (right - left) / 2 (this avoids integer overflow)
   * If the element at mid is equal to the target, return mid
   * If the element at mid is greater than the target, update right = mid - 1
   * If the element at mid is less than the target, update left = mid + 1
3. If the loop exits without finding the element, return -1 to indicate the element is not present



**Source Code**

|  |
| --- |
| **def binary\_sort(arr, target):**  **low = 0**  **high = len(arr) - 1**  **while low <= high:**  **mid = (low + high) // 2**  **if arr[mid] == target:**  **return mid**  **elif arr[mid] < target:**  **low = mid + 1**  **elif arr[mid] > target:**  **high = mid -1**  **return -1**  **arr = [11,17,18,45,50,71,95]**  **print("Element found at index:",binary\_sort(arr, 50))** |

**Sample Input and Output**

**Input:**

|  |
| --- |
| **11,17,18,45,50,71,95** |

**Searching for element: 50**

**Output:**

|  |
| --- |
| **Element found at index: 4** |

**Problem 5: Finding a Pattern in Text using Pattern Matching Algorithm**

**Problem Illustration and Logic**

Pattern matching is the process of finding occurrences of a pattern string within a larger text string. The naive pattern matching algorithm compares the pattern with the text character by character at each possible position until either a match is found or all possibilities are exhausted.

The logic behind the naive pattern matching algorithm is:

1. Slide the pattern over the text one character at a time
2. For each position, compare the pattern with the current substring of the text
3. If all characters match, we have found an occurrence of the pattern
4. Continue searching until we've checked all possible positions in the text

This simple approach checks all potential matches and has a worst-case time complexity of O((n-m+1)\*m), where n is the length of the text and m is the length of the pattern.

**Step-by-step Algorithm**

1. Start at the first character of the text (position 0)
2. Compare each character of the pattern with the corresponding character in the text
3. If all characters match, record the current starting position as a match
4. Slide the pattern one position to the right
5. Repeat steps 2-4 until the end of the text is reached
6. Return all positions where matches were found

**Source Code**

|  |
| --- |
| **def pattern(text, case):**  **n, m = len(text), len(case)**  **for i in range(n-m+1):**  **if text[i:i+m] == case:**  **return i**  **return -1**  **text = "this is a simple example"**  **case = "simple"**  **print("Pattern found at index:", pattern(text, case))** |

**Sample Input and Output**

Input:

|  |
| --- |
| **this is a simple example** |

Output:

|  |
| --- |
| Pattern found at index: 10 |

**Problem 6: Implementation of Queue Data Structure and its Operations**

**Problem Illustration and Logic**

A queue is a linear data structure that follows the **First In, First Out (FIFO)** principle, meaning the element added first is removed first. This implementation uses a Python list to simulate a queue with basic operations: enqueue (insertion), dequeue (removal), peek (viewing the front element), and checking if the queue is empty.

**Step-by-step Algorithm**

1. Create a Queue class with an empty list to store elements.
2. Implement the enqueue(item) method to add an element to the queue.
3. Implement the dequeue() method to remove and return the front element.
4. Implement the is\_empty() method to check if the queue is empty.
5. Implement the peek() method to return the front element without removing it.
6. Implement the display() method to show all elements in the queue.
7. Demonstrate the queue operations by enqueuing and dequeuing elements.

|  |
| --- |
| 1. class Queue: 2. def \_\_init\_\_(self): 3. self.queue = [] 4. def enqueue(self, item): 5. self.queue.append(item) 6. def dequeue(self): 7. if not self.is\_empty(): 8. return self.queue.pop(0) 9. return "Queue is empty" 10. def is\_empty(self): 11. return len(self.queue) == 0 12. def peek(self): 13. if not self.is\_empty(): 14. return self.queue[0] 15. return "Queue is empty" 16. def display(self): 17. return self.queue 18. # Example 19. q = Queue() 20. q.enqueue(10) 21. q.enqueue(20) 22. q.enqueue(30) 23. print("Queue after enqueue:", q.display()) 24. print("Dequeued element:", q.dequeue()) 25. print("Queue after dequeue:", q.display()) |

**Sample Input and Output**

**Output**

|  |
| --- |
| **Queue after enqueue: [10, 20, 30]**  **Dequeued element: 10**  **Queue after dequeue: [20, 30]** |

**Problem 7: N-Queens Problem Using Backtracking**

**Problem Illustration and Logic**

The N-Queens problem is a classic combinatorial problem where we place **N queens** on an **N × N** chessboard such that no two queens attack each other. This means:

* No two queens share the same row.
* No two queens share the same column.
* No two queens share the same diagonal.

Backtracking is used to explore possible placements of queens and backtrack whenever a conflict is found.

**Step-by-step Algorithm**

1. Start with an empty **N × N** chessboard.
2. Place a queen in the first row.
3. Move to the next row and attempt to place a queen in a safe column.
4. If a safe position is found, place the queen and move to the next row.
5. If no safe position is available, backtrack to the previous row and change the column position of the last placed queen.
6. Repeat this process until all queens are placed or all possibilities are exhausted.

**Source Code**

|  |
| --- |
| **N = 8**  **def print\_solution(board):**  **for row in board:**  **print(" ".join("Q" if col else "-" for col in row))**  **print()**  **def is\_safe(board, row, col):**  **for i in range(row):**  **if board[i][col]:**  **return False**    **for i, j in zip(range(row, -1, -1), range(col, -1, -1)):**  **if board[i][j]:**  **return False**    **for i, j in zip(range(row, -1, -1), range(col, N)):**  **if board[i][j]:**  **return False**    **return True**  **def solve\_n\_queens(board, row):**  **if row >= N:**  **print\_solution(board)**  **return True**    **res = False**  **for col in range(N):**  **if is\_safe(board, row, col):**  **board[row][col] = 1**  **res = solve\_n\_queens(board, row + 1) or res**  **board[row][col] = 0 # Backtrack**    **return res**  **def solve():**  **board = [[0] \* N for \_ in range(N)]**  **if not solve\_n\_queens(board, 0):**  **print("No solution exists")**  **solve()** |

**Sample Input and Output:**

**Input**

|  |
| --- |
| **N = 4** |

**Output**

|  |
| --- |
| **- Q - -**  **- - - Q**  **Q - - -**  **- - Q -**  **- - Q -**  **Q - - -**  **- - - Q**  **- Q - -** |

**Problem 8: Sum of Subset Problem Using Backtracking**

**Problem Illustration and Logic**

The **Sum of Subset** problem is a combinatorial problem where we find all subsets of a given set whose sum is equal to a given target value. The problem can be solved using backtracking by exploring all possible subsets and pruning paths that exceed the target sum.

Given:

* **Set S** = {5, 10, 12, 13, 15, 18}
* **Target sum (d)** = 30

**Step-by-Step Algorithm:**

1. Start with an empty subset and a sum of 0.
2. Recursively explore all elements, adding them to the subset if the sum remains ≤ d.
3. If a subset's sum equals **d**, print the subset.
4. If the sum exceeds **d**, backtrack and explore other possibilities.
5. Continue until all subsets are explored.

**Source Code**

|  |
| --- |
| **def sum\_of\_subsets(S, subset, index, current\_sum, target):**  **if current\_sum == target:**  **print(subset)**  **return**    **if index >= len(S) or current\_sum > target:**  **return**    **# Include the current element**  **sum\_of\_subsets(S, subset + [S[index]], index + 1, current\_sum + S[index], target)**    **# Exclude the current element and move to the next**  **sum\_of\_subsets(S, subset, index + 1, current\_sum, target)**  **# Example execution**  **S = [5, 10, 12, 13, 15, 18]**  **target\_sum = 30**  **print("Subsets with sum 30:")**  **sum\_of\_subsets(S, [], 0, 0, target\_sum)** |

**Sample Input and Output:**

**Input**

|  |
| --- |
| **S = [5, 10, 12, 13, 15, 18]**  **target\_sum = 30** |

**Output**

|  |
| --- |
| **[5, 10, 15]**  **[5, 12, 13]**  **[12, 18]** |

**Problem 9: 0/1 Knapsack Problem Using Dynamic Programming**

**Problem Illustration and Logic**

The **0/1 Knapsack Problem** is a classic optimization problem where we aim to maximize the total profit while ensuring that the total weight of selected items does not exceed the knapsack capacity. The dynamic programming approach is used to efficiently solve the problem by storing intermediate results in a table.

Given:

* **Profits (P)** = {15, 25, 13, 23}
* **Weights (W)** = {2, 6, 12, 9}
* **Knapsack Capacity (C)** = 20
* **Number of items (n)** = 4

**Step-by-Step Algorithm:**

1. Create a **dp table** where dp[i][w] represents the maximum profit using the first i items with a knapsack of weight w.
2. Initialize the first row and column to zero (since no items or zero capacity means zero profit).
3. For each item i:
   * If W[i-1] ≤ w, decide whether to include it (P[i-1] + dp[i-1][w-W[i-1]]) or exclude it (dp[i-1][w]).
   * Store the maximum value.
4. The final answer is found at dp[n][C], representing the maximum profit possible.

**Source Code:**

|  |
| --- |
| def knapsack(P, W, C, n):  dp = [[0] \* (C + 1) for \_ in range(n + 1)]    for i in range(1, n + 1):  for w in range(1, C + 1):  if W[i - 1] <= w:  dp[i][w] = max(P[i - 1] + dp[i - 1][w - W[i - 1]], dp[i - 1][w])  else:  dp[i][w] = dp[i - 1][w]    return dp[n][C]  P = [15, 25, 13, 23]  W = [2, 6, 12, 9]  C = 20  n = len(P)  print("Maximum profit:", knapsack(P, W, C, n)) |

**Sample Input and Output:**

**Input**

|  |
| --- |
| **P = [15, 25, 13, 23]**  **W = [2, 6, 12, 9]**  **C = 20**  **n = 4** |

**Output**

|  |
| --- |
| **Maximum profit: 40** |

**Problem 10: Tower of Hanoi Problem Using Recursion**

**Problem Illustration and Logic**

The **Tower of Hanoi** is a mathematical puzzle that consists of three rods and N disks of different sizes. The objective is to move all disks from the source rod to the destination rod using an auxiliary rod, following these rules:

1. Only one disk can be moved at a time.
2. A larger disk cannot be placed on a smaller disk.
3. All disks must be moved from the source to the destination rod while following the above constraints.

The problem can be solved recursively using the following approach:

1. Move the top **N-1** disks from the source rod to the auxiliary rod.
2. Move the **Nth (largest) disk** from the source rod to the destination rod.
3. Move the **N-1** disks from the auxiliary rod to the destination rod.

**Step-by-Step Algorithm:**

1. Define a recursive function to move N disks from the source to the destination rod using an auxiliary rod.
2. If **N == 1**, directly move the disk to the destination rod.
3. Recursively move **N-1** disks to the auxiliary rod, move the **Nth** disk to the destination, and then move **N-1** disks from the auxiliary rod to the destination.
4. Print the steps of the moves.

**Source Code:**

|  |
| --- |
| def tower\_of\_hanoi(n, source, auxiliary, destination):  if n == 1:  print(f"Move disk 1 from {source} to {destination}")  return    tower\_of\_hanoi(n - 1, source, destination, auxiliary)  print(f"Move disk {n} from {source} to {destination}")  tower\_of\_hanoi(n - 1, auxiliary, source, destination)  # Example execution  N = 3 # Change N for different disk numbers  print(f"Solution for {N} disks:")  tower\_of\_hanoi(N, 'A', 'B', 'C') |

**Sample Input and Output:**

**Input**

|  |
| --- |
| **N = 3** |

**Output**

|  |
| --- |
| **Solution for 3 disks:**  **Move disk 1 from A to C**  **Move disk 2 from A to B**  **Move disk 1 from C to B**  **Move disk 3 from A to C**  **Move disk 1 from B to A**  **Move disk 2 from B to C**  **Move disk 1 from A to C** |