LAB REPORT



Department of

Information and Communication Engineering

ICE - 2202

Data Structure and Algorithm Sessional

Submitted By Azmira Akter Simla Roll: 220626

Dept. of Information and Communication Engineering

Submitted To Md. Anwar Hossain Professor

Dept. of Information and

Communication Engineering

Date of Submission: 01-03-25

**Index**

,

|  |  |
| --- | --- |
| **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. |

# Problem Number: 1

**Title:** Write a program to sort a linear array using the bubble sort algorithm.

# Illustration of the Problem:

The bubble sort algorithm works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items, and swapping them if they are in the wrong order. This process is repeated until the list is sorted.

# Step-by- Step Procedure (Algorithm):

1. Step-1: Get the length of the array.
2. Step-2: Run a loop from i = 0 to n-1 representing the number of passes through the array.
3. Step-3: Inside the outer loop, run another loop from j = 0 to n-i-2 to ensure the last i elements are already sorted.
4. Step-4: Compare adjacent elements arr[j] and arr[j+1]. If arr[j] > arr[j+1], swap them.
5. Step-5: Continue this process until no swaps occur, indicating the array is sorted.

# Source Code (C++):

**//This program is written by AZMIRA AKTER SIMLA, Roll: 220626,**

**Session:2021-2022**

#include <iostream> using namespace std;

int main() {

int n;

cout << "Enter the number of elements in the array: "; cin >> n;

int arr[n];

cout << "Enter the elements of the array: "; for (int i = 0; i < n; i++) {

cin >> arr[i];

}

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) { if (arr[j] > arr[j + 1]) {

int temp = arr[j]; arr[j] = arr[j + 1]; arr[j + 1] = temp;

}

}

}

cout << "Sorted array in ascending order: "; for (int i = 0; i < n; i++) {

cout << arr[i] << " ";

}

return 0;

}

# Input:

Enter the number of elements in the array: 5 Enter the elements of the array: 6 4 9 2 0

**Output:** Sorted array in ascending order: 0 2 4 6 9

# Problem Number: 2

**Title:** Write a program to find an element using a linear search algorithm.

# Illustration of the Problem:

Linear Search is a simple searching algorithm that checks each element in a list one by one until the desired element is found or the entire list is traversed. This method is useful for unsorted lists where elements do not follow a specific order.

# Step-by-Step Procedure (Algorithm):

1. Step-1: Define the array arr\_list and the target element key. Accept user input for the number of elements and the space-separated list of elements.
2. Step-2: Define a function count\_elements() that counts the number of elements in the input array by iterating over it.
3. Step-3: Start a loop from i = 0 to the total number of elements in the array.
4. Step-4: At each iteration, compare arr[i] with key. If they are equal, return i, indicating the position where the element is found.
5. Step-5: If no match is found after checking all the elements, return -1, meaning the element is not present in the array.
6. Step-6: Display the result. If the index is not -1, print the position; otherwise, print that the element was not found.

# Source Code (C++):

**//This program is written by AZMIRA AKTER SIMLA, Roll: 220626,**

**Session:2021-2022**

#include <iostream> using namespace std;

int main() {

int n, key, found = 0;

cout << "Enter the number of elements in the array: "; cin >> n;

int arr[n];

cout << "Enter the elements of the array: "; for (int i = 0; i < n; i++) {

cin >> arr[i];

}

cout << "Enter the element to search: "; cin >> key;

for (int i = 0; i < n; i++) { if (arr[i] == key) {

cout << "Element " << key << " found at index " << i << "." << endl;

found = 1; break;

}

}

if (!found) {

cout << "Element " << key << " not found in the array." << endl;

}

return 0;

}

# Input:

Enter the number of elements in the array: 5 Enter the elements of the array: 6 8 4 2 7 Enter the element to search: 2

# Output:

Element 2 found at index 3.

# Problem Number:03

**Problem Title:** Write a program to sort a linear array using the merge sort algorithm.

# Illustration of the Problem:

Merge Sort is a sorting technique that follows the divide-and-conquer approach. The process can be broken down into three main steps:

1. Dividing the Array:
   * The given array is divided into two smaller sub-arrays (left and right) until each sub-array contains only one element.
   * This is done recursively, meaning each sub-array is further split into halves until reaching single-element sub-arrays.
2. Sorting the Sub-arrays:
   * Once the array is broken down into individual elements, the merging process begins.
   * Two adjacent sub-arrays are compared element by element and arranged in ascending order.
3. Merging the Sorted Sub-arrays:
   * The two sorted sub-arrays are then combined to form a larger sorted array.
   * This merging process continues recursively until the entire array is sorted.

This approach ensures that the array is efficiently sorted with a time complexity of O(n log n).

# Step-by-Step Procedure:

1. Step-1: Define the merge\_sort function that takes an array as input.
2. Step-2: Check if the array has more than one element; if not, return the array (if len(arr) <= 1: return arr).
3. Step-3: Divide the array into left and right halves without using built-in functions (mid = len(arr) // 2, left = [arr[i] for i in range(mid)], right = [arr[i] for i in range(mid, len(arr))]).
4. Step-4: Recursively call merge\_sort on both halves (sorted\_left = merge\_sort(left), sorted\_right = merge\_sort(right)).
5. Step-5: Define a merge function to combine two sorted halves without using built-in functions (merge(sorted\_left, sorted\_right)).
6. Step-6: Use simple loops and conditions to merge elements in sorted order (while i < len(left) and j < len(right): if left[i] < right[j]: result[k] = left[i]; i

+= 1 else: result[k] = right[j]; j += 1).

1. Step-7: Return the merged sorted array (return result).
2. Step-8: Test the algorithm with a sample array (arr = [38, 27, 43, 3, 9, 82,

10]).

1. Step-9: Print the sorted output (print("Sorted array:", sorted\_arr)).

# Source Code (C++):

**//This program is written by AZMIRA AKTER SIMLA, Roll: 220626,**

**Session:2021-2022**

#include <iostream> using namespace std;

void mergeSort(int arr[], int left, int right) { if (left >= right) return;

int mid = left + (right - left) / 2; mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

int n1 = mid - left + 1, n2 = right - mid; int leftArr[n1], rightArr[n2];

for (int i = 0; i < n1; i++) leftArr[i] = arr[left + i];

for (int j = 0; j < n2; j++) rightArr[j] = arr[mid + 1 + j];

int i = 0, j = 0, k = left; while (i < n1 && j < n2) {

arr[k++] = (leftArr[i] <= rightArr[j]) ? leftArr[i++] : rightArr[j++];

}

while (i < n1) arr[k++] = leftArr[i++]; while (j < n2) arr[k++] = rightArr[j++];

}

int main() { int n;

cout << "Enter the number of elements in the array: "; cin >> n;

int arr[n];

cout << "Enter the elements of the array: "; for (int i = 0; i < n; i++) {

cin >> arr[i];

}

mergeSort(arr, 0, n - 1);

cout << "The sorted array is: "; for (int i = 0; i < n; i++) {

cout << arr[i] << " ";

}

cout << endl;

return 0;

}

# Input:

Enter the number of elements in the array: 7

Enter the elements of the array: 38 27 43 3 9 82 10

# Output:

The sorted array is: 3 9 10 27 38 43 82

# Problem Number:04

**Problem Title:** Write a program to find an element using the binary search algorithm.

# Illustration of the Problem:

To find an element in a sorted list, the binary search algorithm works by repeatedly dividing the list in half. This method eliminates half of the remaining elements at each step, drastically reducing the number of comparisons compared to a linear search.

The process begins by checking the middle element of the list. If this element is the target, the search is over. If the middle element is larger than the target, the target must lie in the left half of the list. Conversely, if the middle element is smaller than the target, the search continues in the right half of the list.

This "divide and conquer" approach continues until the element is found or the search space is exhausted. The formula used to calculate the middle index is:

Middle = 𝑙𝑜𝑤 + ℎ𝑖𝑔ℎ

2

where low is the starting index and high is the ending index of the list.

# Step-by-Step Procedure (Algorithm):

1. Step-1: Initialize an empty list arr.
2. Step-2: Take input of space-separated numbers as a string.
3. Step-3: Convert the string into a list of integers by accumulating digits and adding them to arr.
4. Step-4: Take the target number as input.
5. Step-5: Set low = 0 and high = len(arr) - 1.
6. Step-6: While low <= high:
   * Calculate middle = (low + high) // 2.
   * If arr[middle] == target, return middle.
   * If arr[middle] > target, set high = middle - 1.
   * If arr[middle] < target, set low = middle + 1.
7. Step-7: If the element is not found, return -1.
8. Step-8: Output the index or "Element not found".

# Source Code (C++):

**//This program is written by AZMIRA AKTER SIMLA, Roll: 220626,**

**Session:2021-2022**

#include <iostream> using namespace std;

int main() {

int n, target, left, right, mid, step = 1;

// Taking array size

cout << "Enter the number of elements: "; cin >> n;

int arr[n];

// Taking sorted array

cout << "Enter " << n << " sorted elements: "; for (int i = 0; i < n; i++) {

cin >> arr[i];

}

// Searching number

cout << "Enter the number to search: "; cin >> target;

// Binary Search Logic left = 0; right = n - 1; while (left <= right) {

mid = left + (right - left) / 2;

// Printing the position if (arr[mid] == target) {

cout << "Target " << target << " found at position " << mid + 1 << endl; return 0;

} else if (arr[mid] < target) { left = mid + 1;

} else {

right = mid - 1;

}

step++;

}

cout << "Target " << target << " not found in the array" << endl; return 0;

}

# Input:

Enter the number of elements: 5 Enter 5 sorted elements: 3 4 5 6 9 Enter the number to search: 9

# Output:

Target 9 found at position 5

# Problem Number: 05

**Problem Title:** Write a Program to Find a Given Pattern from Text Using the Pattern Matching Algorithm.

# Illustration of the Program:

To find a pattern in a text, we check if the pattern exists at any position in the text. The basic idea is to compare the pattern with substrings of the text, one by one, until a match is found or the whole text is searched.

# Step-by-Step Procedure:

1. Step-1: Initialize variables:
   * text (input string to search in)
   * pattern (input string to find)
2. Step-2: Set n = len(text) and m = len(pattern).
3. Step-3: Loop through the text from index 0 to n - m:
   * Compare the substring of text[i:i+m] with pattern.
   * If they match, return the index i (position where the pattern starts).
4. Step-4: If no match is found after checking all possible substrings, return -1.

# Source Code (C++):

**//This program is written by AZMIRA AKTER SIMLA, Roll: 220626,**

**Session:2021-2022**

#include <iostream> #include <cstring> using namespace std;

int main() {

char text[1000], pattern[1000];

// Taking input for the text and pattern cout << "Enter the text: "; cin.getline(text, sizeof(text));

cout << "Enter the pattern: "; cin.getline(pattern, sizeof(pattern));

int textLen = strlen(text);

int patternLen = strlen(pattern);

// Loop through the text to find the pattern

for (int i = 0; i <= textLen - patternLen; i++) { int j = 0;

// Check for matching characters in the text and pattern while (j < patternLen && text[i + j] == pattern[j]) {

j++;

}

// If a match is found if (j == patternLen) {

cout << "Pattern found at index " << i << endl; return 0;

}

}

cout << "Pattern not found in the text" << endl; return 0;

}

# Input:

Enter the text: Information and Communication Engineering Enter the pattern: Communication

# Output:

Pattern found at index 16

# Problem Number: 06

**Problem Title**: Implement a Queue Data Structure and Perform Typical Operations Such as Enqueue, Dequeue, and Checking if the Queue is Empty.

# Illustration of the Program:

A queue is a data structure that follows the FIFO (First In, First Out) principle. In a queue, the first element added is the first one to be removed. To implement a queue, we manage two main operations: enqueue and dequeue. When we enqueue, we add an element to the rear of the queue, and when we dequeue, we remove the element from the front.

The queue can be visualized as a list where elements are stored sequentially. The element at the front of the list is the first one to be removed, while the element at the rear is the last one to be added. Once an element is added to the queue, it remains there until it reaches the front position to be removed. If we want to access or remove an element from the front of the queue, we need to shift the elements so that the next one in line moves to the front. This is done until the queue is empty.

# Step-by-Step Procedure:

1. Step-1: Create a Queue class with an empty list to store elements.
2. Step-2: Define the enqueue method to add an element to the end of the list.
3. Step-3: Define the dequeue method to remove and return the first element.
4. Step-4: Define the front method to return the first element without removing it.
5. Step-5: Define the is\_empty method to check if the list is empty.
6. Step-6: Prompt the user to enter the number of elements they want to enqueue.
7. Step-7: Use a loop to take input from the user for each element and enqueue it.
8. Step-8: After user input, display the front element of the queue.
9. Step-9: Perform the dequeue operation and display the dequeued element.
10. Step-10: Check and display whether the queue is empty.
11. Step-11: Display the front element of the queue again after the dequeue operation.

# Source Code (C++):

//This program is written by AZMIRA AKTER SIMLA, Roll: 220626,

Session:2021-2022

#include <iostream> using namespace std;

#define MAX 5

int queue[MAX], front = -1, rear = -1;

int main() {

int choice, value;

while (1) {

cout << "\nQueue Operations:\n"; cout << "1. Enqueue\n";

cout << "2. Dequeue\n"; cout << "3. Display\n"; cout << "4. Exit\n";

cout << "Enter your choice: "; cin >> choice;

switch (choice) { case 1:

if (rear == MAX - 1) {

cout << "Queue is full! Cannot enqueue.\n";

} else {

if (front == -1) { front = 0;

}

cout << "Enter the value to enqueue: "; cin >> value;

rear++;

queue[rear] = value;

cout << value << " enqueued to the queue.\n";

cout << "Current front: " << front << ", rear: " << rear << endl;

}

break;

case 2:

if (front == -1 || front > rear) {

cout << "Queue is empty! Cannot dequeue.\n";

} else {

cout << "Dequeued value: " << queue[front] << endl; front++;

cout << "Current front: " << front << ", rear: " << rear << endl;

if (front > rear) { front = rear = -1;

cout << "Queue is now empty.\n";

}

}

break;

case 3:

if (front == -1 || front > rear) { cout << "Queue is empty!\n";

} else {

cout << "Queue elements are: "; for (int i = front; i <= rear; i++) {

cout << queue[i] << " ";

}

cout << endl;

}

break;

case 4:

cout << "Exiting...\n"; return 0;

default:

cout << "Invalid choice! Try again.\n";

}

if (front == -1 || front > rear) { cout << "Queue is empty.\n";

} else {

cout << "Current queue: ";

for (int i = front; i <= rear; i++) { cout << queue[i] << " ";

}

cout << endl;

}

}

return 0;

}

# Input:

Enter the number of elements to enqueue: 3 Enter element to enqueue: 10

Enter element to enqueue: 20 Enter element to enqueue: 30

# Output:

Queue Operations:

1. Enqueue
2. Dequeue
3. Display
4. Exit

Enter your choice: 1

10 enqueued to the queue. Current front: 0, rear: 0

Queue Operations:

1. Enqueue
2. Dequeue
3. Display
4. Exit

Enter your choice: 1

20 enqueued to the queue. Current front: 0, rear: 1

Queue Operations:

1. Enqueue
2. Dequeue
3. Display
4. Exit

Enter your choice: 2 Dequeued value: 10

Current front: 1, rear: 1 Queue is now empty.

# Problem Number: 07

**Problem Title:** Solving the N Queen's Problem Using Backtracking.

# Illustration of the Program:

The N Queen's problem involves placing N queens on an N x N chessboard such that no two queens can attack each other. A queen can attack any other piece in the same row, column, or diagonal. The goal is to find a way to arrange all N queens on the board, ensuring that no two queens share the same row, column, or diagonal. To solve this problem, backtracking is used. Backtracking is a systematic method of trying possible solutions and eliminating those that fail. The process begins by attempting to place a queen in a specific position on the board. Then, the algorithm moves to the next row and tries to place a queen in a safe position. If placing a queen’s results in a conflict (i.e., another queen can attack it), the algorithm backtracks and attempts a different position for the previously placed queen.

The solution is found by exploring all possible placements for the queens, ensuring that each queen is placed safely before moving on to the next row. If no valid positions exist for a queen in a given row, the algorithm returns to the previous row to explore other potential positions. This method continues until all N queens are placed on the board in valid positions, or all possible configurations are exhausted.

# Step-by-Step Procedure:

* Step-1: Initialize an array to store the column positions of queens, where each index represents a row and the value at each index represents the column.
* Step-2: Define the isSafe method to check if a queen can be placed in a particular position without conflicting with other queens (no two queens share the same column or diagonal).
* Step-3: Use recursion in the solveNQueens method to try placing queens in each row.
* Step-4: If a queen is successfully placed in a row, move on to the next row.
* Step-5: If placing a queen’s results in no safe position for any column in a row, backtrack and move the queen in the previous row to another column.
* Step-6: Once all queens are placed safely, print the board solution.
* Step-7: If no solution is found after exploring all configurations, indicate that no solution exists.

# Source Code (C++):

//This program is written by AZMIRA AKTER SIMLA, Roll: 220626,

Session:2021-2022

#include <iostream> using namespace std;

#define N 4

void printSolution(int placed[]) { static int solutionCount = 0;

cout << "\nSolution " << ++solutionCount << ":\n"; for (int i = 0; i < N; i++, cout << "\n")

for (int j = 0; j < N; j++)

cout << (placed[i] == j ? 1 : 0) << " ";

}

bool isSafe(int placed[], int row, int col) { for (int prev = 0; prev < row; prev++) {

if (placed[prev] == col || placed[prev] - prev == col - row ||

placed[prev] + prev == col + row) { return false;

}

}

return true;

}

void solveNQueens(int placed[], int row) { if (row == N) {

printSolution(placed); return;

}

for (int col = 0; col < N; col++) { if (isSafe(placed, row, col)) {

placed[row] = col; solveNQueens(placed, row + 1);

}

}

}

int main() {

int placed[N] = {-1}; solveNQueens(placed, 0);

return 0;

}

# Input:

Enter the number of queens: 4

# Output:

Solution 1:

1 0 0 0

0 0 1 0

0 1 0 0

0 0 0 1

Solution 2:

0 0 0 1

1 0 0 0

0 1 0 0

0 0 1 0

# Problem Number: 08

**Problem Title:** To Implement the Sum of Subsets Problem Using Backtracking

# Illustration of the Program:

In this problem, we are given a set S= {5,10,12,13,15,18} and a target sum d=30. The goal is to find a subset of SSS such that the sum of the elements in that subset is equal to d. The subset sum problem can be solved by exploring all possible

subsets of SSS and checking if any of them add up to the target sum.

To solve this problem, we use a bit masking approach, which iterates through all possible subsets and computes the sum of elements for each subset. If the sum of any subset equals d, it is a valid solution.

For example, consider the set S= {5,10,12,13,15,18} and the target sum d = 30. The process begins by trying all possible subsets and calculating their sum. When the subset {5,10,15} is found, it adds up to 30, which matches the target sum.

Thus, the subset {5,10,15 is a valid solution.

# Step-by-Step Procedure:

* Step-1: Initialize an array S[]with the elements of the set.
* Step-2: Get the number of elements N and the target sum d.
* Step-3: Iterate over all possible subsets using bit masking (from 0 to 2^N - 1).
* Step-4: For each subset, calculate the sum of the elements included in that subset.
* Step-5: If the sum of the subset equals d, print the subset.
* Step-6: If no subset is found after iterating through all possibilities, output "No subset with sum d".

# Source Code (C++):

//This program is written by AZMIRA AKTER SIMLA, Roll: 220626,

Session:2021-2022

#include <iostream> #include <cmath> using namespace std;

int main() {

int N, target\_sum;

cout << "Enter the number of elements: "; cin >> N;

int S[N];

cout << "Enter the elements: "; for (int i = 0; i < N; i++) {

cin >> S[i];

}

cout << "Enter the target sum: "; cin >> target\_sum;

int total\_subsets = 1 << N; int count = 0;

for (int mask = 0; mask < total\_subsets; mask++) { int subset\_sum = 0;

bool found = false;

for (int j = 0; j < N; j++) { if (mask & (1 << j)) {

subset\_sum += S[j];

}

}

if (subset\_sum == target\_sum) { found = true;

cout << "{ ";

for (int j = 0; j < N; j++) { if (mask & (1 << j)) {

cout << S[j] << " ";

}

}

cout << "}\n"; count++;

}

}

cout << "Total subsets found: " << count << endl;

return 0;

}

# Input:

Enter the number of elements: 6 Enter the elements: 5 10 12 13 15 18 Enter the target sum: 30

# Output:

{5 10 15}

Total subsets found: 1

# Problem Number: 09

**Problem Title:** To Implement the 0/1 Knapsack Problem Using Dynamic Programming.

# Illustration of the Problem:

In the 0/1 Knapsack problem, we are given a set of items, each with a profit and a weight, and a knapsack with a fixed capacity. The objective is to select a combination of items that maximizes the total profit while ensuring that their total weight does not exceed the knapsack’s capacity.

Given:

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

The goal is to determine the most profitable combination of items to include in the knapsack, where we either include or exclude each item, ensuring the total weight does not exceed the capacity. Dynamic programming helps solve this by building a table to keep track of the maximum profit for each item and knapsack capacity, ultimately providing the optimal solution.

# Step-by-Step Procedure:

1. Step-1: Initialize a 2D array dp with dimensions (n+1) × (C+1), where n is the number of items and C is the knapsack capacity. Set all values in the table to 0.
2. Step-2: For each item from 1 to n (denoted as i), perform the following:
3. Step-3: For each possible capacity from 1 to C (denoted as w), perform the following:
4. Step-4: If the current item's weight W[i-1] is less than or equal to the current capacity w:
   * Calculate the profit if the item is included: dp[i][w] = dp[i-1][w-W[i- 1]] + P[i-1]
   * Calculate the profit if the item is excluded: dp[i][w] = dp[i-1][w]
   * Store the maximum of these two values in dp[i][w].
5. Step-5: If the current item's weight W[i-1] is greater than the current capacity w, exclude the item: dp[i][w] = dp[i-1][w]
6. Step-6: After filling the entire table, the value at dp[n][C] will contain the maximum profit that can be obtained given the knapsack capacity and available items.

# Source Code (C++):

//This program is written by AZMIRA AKTER SIMLA, Roll: 220626,

Session:2021-2022

#include <iostream> using namespace std;

int main() {

int profits[] = {15, 25, 13, 23};

int weights[] = {2, 6, 12, 9};

int n = sizeof(profits) / sizeof(profits[0]); int capacity = 20;

int dp[n+1][capacity+1];

// Initialize the dp table

for (int i = 0; i <= n; i++) {

for (int w = 0; w <= capacity; w++) {

if (i == 0 || w == 0) { dp[i][w] = 0;

} else if (weights[i-1] <= w) {

dp[i][w] = max(dp[i-1][w], profits[i-1] + dp[i-1][w-weights[i-1]]);

} else {

dp[i][w] = dp[i-1][w];

}

}

}

// Output the result

cout << "The maximum profit that can be obtained is: " << dp[n][capacity] << endl;

return 0;

}

# Input:

* Profits (P): P= (15,25,13,23)
* Weights (W): W= (2,6,12,9)
* Knapsack Capacity (C): C=20

# Output:

The maximum profit that can be obtained is: 38

# Problem Number: 10

**Problem Title:** To Implement a Program to Solve the Tower of Hanoi Problem for N Disks.

# Illustration of the Problem:

The Tower of Hanoi is a mathematical puzzle that involves three pegs (source, destination, and auxiliary) and N disks of different sizes. The objective is to move all disks from the source peg to the destination peg following these rules:

* Only one disk can be moved at a time.
* A larger disk cannot be placed on top of a smaller disk.
* Disks can only be moved between the three pegs.

# Step-by-Step Procedure:

1. Step-1: Take user input for the number of disks (N), source peg, destination peg, and auxiliary peg.
2. Step-2: Calculate the total number of moves required using the formula: Total Moves = (2^N) - 1.
3. Step-3: Initialize a stack to store the state of moves.
4. Step-4: Push the initial state onto the stack with N, source peg, destination peg, and auxiliary peg.
5. Step-5: While the stack is not empty, perform the following steps:
   * If N == 1, print the move from the source peg to the destination peg.
   * Else, push three states onto the stack in the following order:
     + Move N-1 disks from auxiliary to destination using the source peg.
     + Move the largest disk from source to destination.
     + Move N-1 disks from source to auxiliary using the destination peg.
6. Step-6: Repeat the process until all moves are executed and printed.

# Source Code (C++):

**//This program is written by AZMIRA AKTER SIMLA, Roll: 220626,**

**Session:2021-2022**

#include <iostream> using namespace std;

int main() { int num;

char source, destination, auxiliary;

// Prompting user for input

cout << "Enter the number of disks: "; cin >> num;

cout << "Enter the source peg, destination peg, and auxiliary peg: "; cin >> source >> destination >> auxiliary;

int total\_moves = (1 << num) - 1; // Calculate total moves cout << "Total number of moves: " << total\_moves << endl;

int stack[1000][4]; int top = -1;

stack[++top][0] = num; stack[top][1] = source; stack[top][2] = destination; stack[top][3] = auxiliary;

while (top >= 0) {

int n = stack[top][0];

char from\_peg = stack[top][1]; char to\_peg = stack[top][2]; char aux\_peg = stack[top--][3];

if (n == 1) {

cout << from\_peg << " -> " << to\_peg << endl;

} else {

stack[++top][0] = n - 1; stack[top][1] = aux\_peg; stack[top][2] = to\_peg; stack[top][3] = from\_peg;

stack[++top][0] = 1; stack[top][1] = from\_peg; stack[top][2] = to\_peg; stack[top][3] = aux\_peg;

stack[++top][0] = n - 1; stack[top][1] = from\_peg; stack[top][2] = aux\_peg; stack[top][3] = to\_peg;

}

}

return 0;

}

# Input:

Enter the number of disks: 3

Enter the source peg, destination peg, and auxiliary peg: A C B

# Output:

Total number of moves: 7 A -> C

A -> B C -> B

A -> C

B -> A B -> C A -> C