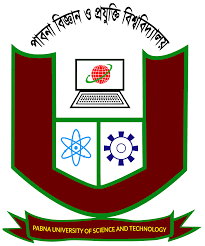
**PABNA UNIVERSITY OF SCIENCE AND TECHNOLOGY**

****

**Faculty of Engineering and Technology**

***Department Of***

***Information and Communication Engineering***

**Lab Report**

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

**Course Code: ICE-2202**

**Submitted By:**

**Submitted To:**

**Md. Anwar Hossain**

**Chairman**

**Department of ICE, Pabna University of Science and Technology**

**Name: Mst. Sabi Afrin Mim**

**Roll: 220615**

**Session: 2021-2022**

**2nd Year 2nd Semester**

**Department of ICE, PUST**

**INDEX**

|  |  |
| --- | --- |
| **Experiment No** | **Experiment Name** |
| **01** | **Implementation of Bubble Sort Algorithm** |
| **02** | **Find an element using linear search algorithm.** |
| **03** | **Merge Sort Algorithm Implementation and Analysis** |
| **04** | **Find an element using binary search algorithm.** |
| **05** | **Implementation of pattern matching algorithm.** |
| **06** | **Implement a queue data structure along with its typical operations.** |
| **07** | **Solving N-Queens Problem Efficiently Using Backtracking Algorithm.** |
| **08** | **Subset Sum Problem Using Backtracking** |
| **09** | **Solving the 0/1 Knapsack Problem using Dynamic Programming** |
| **10** | **Solving the Tower of Hanoi Problem using an Iterative Approach with a Stack.** |

**Experiment No: 01**

**Experiment Name: Implementation of Bubble Sort Algorithm in C++.**

**Theory: Bubble Sort** is a simple comparison-based sorting algorithm. It works by repeatedly stepping through the array, comparing adjacent elements, and swapping them if they are in the wrong order. This process is repeated until the array is sorted, with the largest elements "bubbling up" to their correct positions after each pass.

The program takes input from the user for the number of elements and the array, applies the **Bubble Sort** algorithm, and then prints the sorted array.

**Algorithm:**

1. Start
2. Input of the size of the n.
3. Input the element of the array.
4. Bubble Sort:

* Repeat the process n-1 times.
* For each pass, iterate through the array from j = 0 to n-1-i.
* For j = 0 to n-1-I,If arr[j] > arr[j+1] then swap arr[j] and arr[j+1].
* After each pass, the largest element among the unsorted elements moves to the correct position.

1. Print the original array.
2. Print the sorted array after applying Bubble sort.
3. End.

**Source Code:**

#include <iostream>

using namespace std;

int main() {

    int n;

    // Input the size

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

    cin >> n;

    int arr[n];

    // Input the elements

    cout << "Enter the elements of the array: ";

    for (int i = 0; i < n; i++) {

        cin >> arr[i];

    }

    // Bubble Sort logic

    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;

            }

        }

        // Print the array after each pass

        cout << "Array after pass " << i + 1 << ": ";

        for (int k = 0; k < n; k++) {

            cout << arr[k] << " ";

        }

        cout << endl;    }

    // Final sorted array

    cout << "Sorted array in ascending order: ";

    for (int i = 0; i < n; i++) {

        cout << arr[i] << " ";

    }

    cout << endl;

    return 0;

}

**Input:**

Enter the number of element of array: 5

Enter 5 elements: 5 3 8 1 2

**Output:**

**Original array: 5 3 8 1 2**

**Array after pass 1: 3 5 1 2 8**

**Array after pass 2: 3 1 2 5 8**

**Array after pass 3: 1 2 3 5 8**

**Array after pass 4: 1 2 3 5 8**

**Sorted array: 1 2 3 5 8**

**Experiment No: 02**

**Experiment Name: Find an element using linear search algorithm.**

**Theory:**

Linear Search is a basic searching algorithm where each element of the array is checked one by one in sequence until the target element is found or the entire array is traversed. The algorithm works by iterating through the array and comparing each element with the search key If a match is found, the index of the element is returned; otherwise, the algorithm continues until the end of the array and returns a message indicating that the element is not found.

It is a simple and straightforward algorithm that does not require the array to be sorted beforehand. However, its efficiency is reduced when dealing with large arrays since it may require checking every element.

**Algorithm:**

1. Start
2. Initialize the array and the search key.
3. **Set** the variable position to -1, which indicates that the element has not been found initially.
4. Loop through the array:

* For each elements ,compare it with the search key
* If current element equal to the search key, then set the position = i(current index).
* Exit the loop.

1. If the element is not found, continue checking the next element.
2. Check if (position !=-1):

* If true, print the position where the element was found.
* If false, print the element is not found.

1. End.

**Source Code:**

#include <iostream>

using namespace std;

int main() {

    int arr[] = {5, 10, 15, 20, 25, 30, 35};

    int key = 40;

    int position = -1;

       for (int i = 0; i < 7; i++) {

        if (arr[i] == key) {

            position = i;

            break;

        }

    }

    if (position != -1) {

        cout << "Element " << key << " found at position " << position << "." << endl;

    } else {

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

    }

    return 0;

}

**Input:**

The Array: {5 ,10,15,20,25,30,35}

Search key: 40

**Output:**

Element 40 not found in the array

**Experiment No: 03**

**Experiment Name: Merge Sort Algorithm Implementation and Analysis.**

**Theory:**

Merge Sort is a divide-and-conquer based sorting algorithm. It works by recursively dividing the array or list into two halves, sorting those halves individually, and then merging them back together to form a sorted array.

**Steps:**

1. **Divide:** The array is divided into two halves. This process continues recursively until each subarray contains only one element, which is trivially sorted.
2. **Conquer:** Each of the subarrays is sorted independently. After sorting the subarrays, they are merged back into a single sorted array.
3. **Combine:** The sorted subarrays are merged back together into one complete sorted array. This merging process ensures that the array is sorted.

The merge operation is the most crucial part of the algorithm, where two sorted subarrays are combined into one sorted array.

**Algorithm:**

1. Create a temporary array temp[] of size h - l + 1.
2. Initialize i = l, j = mid + 1, and k = 0.
3. Compare elements from both halves (A[i] and A[j]):
   * If A[i] <= A[j], add A[i] to temp[k], increment i.
   * Else, add A[j] to temp[k], increment j.
   * Increment k after each comparison.
4. If any elements remain in the left half (i <= mid), copy them into temp[].
5. If any elements remain in the right half (j <= h), copy them into temp[].
6. Copy all elements from temp[] back to A[] starting at index 1.
7. If l<h

* Find the middle index mid = (l+h)/2
* Recursively sorted the left half: MergeSort(A,l,mid)
* Recursively sort the left half: MergeSort(A,mid+1,h)
* Merge the sorted: Merge(A,l,mid,h)

1. Print sorted Array.
2. End.

**Source code:**

#include <iostream>

using namespace std;

// Merge Function

void merge(int A[], int l, int mid, int h) {

    int i = l, j = mid + 1, k = 0;

    int temp[h - l + 1];

    // Merging the two halves into the temp array

    while (i <= mid && j <= h) {

        if (A[i] <= A[j]) {

            temp[k] = A[i];

            i++;

        } else {

            temp[k] = A[j];

            j++;

        }

        k++;

    }

    // elements in the left half

    while (i <= mid) {

        temp[k] = A[i];

        i++;

        k++;

    }

    // elements in the right half

    while (j <= h) {

        temp[k] = A[j];

        j++;

        k++;

    }

       for (int i = 0; i < k; i++) {

        A[l + i] = temp[i];

    }

}

// Merge Sort Function

void mergeSort(int A[], int l, int h) {

    if (l < h) {

        int mid = (l + h) / 2; // Find the middle index

        mergeSort(A, l, mid);     // Recursively sort the left half

        mergeSort(A, mid + 1, h); // Recursively sort the right half

        merge(A, l, mid, h);      // Merge the two sorted halves

    }

}

// Main Function

int main() {

    int n;

    cout << "Enter the number of elements: ";

    cin >> n;

    int A[n];

    cout << "Enter the elements: ";

    for (int i = 0; i < n; i++) {

        cin >> A[i];

    }

    mergeSort(A, 0, n - 1);

    cout << "Sorted array: ";

    for (int i = 0; i < n; i++) {

        cout << A[i] << " ";    }

    return 0;

}

**Input:**

Enter the number of elements: 5.

Enter the elements: 12 4 6 10 8

**Output:**

Sorted array: 4 6 8 10 12

**Experiment No: 04**

**Experiment Name**: Find an element using binary search algorithm.

**Theory:**

Binary Search is a divide and conquer algorithm used to search for an element in a sorted array or list. The algorithm works by comparing the target element (key) with the middle element of the array:

* If the middle element is equal to the key, the search ends and the index of the middle element is returned.
* If the middle element is smaller than the key, the search continues in the right half of the array.
* If the middle element is greater than the key, the search continues in the left half of the array. The process is repeated until the key is found or the search space is reduced to zero.

**Algorithm:**

 **Start**

* Initialize low to 0 and high to size of the array - 1.

 **Repeat** the following steps while low ≤ high:

* Calculate the middle index: mid = low + (high - low) / 2
* If arr[mid] == key, return mid (Element found).
* If arr[mid] < key, update low = mid + 1 (search the right half).
* Else, update high = mid - 1 (search the left half).

 If key is not found, return -1 (Element not found).

 **End**

**Source Code:**

#include <iostream>

using namespace std;

// Function for Binary Search

int binarySearch(int arr[], int size, int key) {

    int low = 0, high = size - 1;

    while (low <= high) {

        int mid = low + (high - low) / 2;

        if (arr[mid] == key) {

            return mid;

        }

        if (arr[mid] < key) {

            low = mid + 1;

        } else {

            high = mid - 1;

        }

    }

    return -1;

}

int main() {

    int arr[] = {10, 20, 30, 40, 50};  // Sorted array

    int key = 30;

    int result = binarySearch(arr, 5, key);

    if (result != -1) {

        cout << "Element found at index " << result << endl;

    } else {

        cout << "Element not found!" << endl;

    }

    return 0;

}

**Input:**

Array: {10,20,30,40,50)

Search key: 30

**Output:**

Element found at index 2

**Experiment No:** 05

**Experiment Name**: **Implementation of pattern matching algorithm.**

**Theory:**

The pattern matching method involves checking each position of the text string (T) to see if the substring starting at that position matches the pattern string (P). If a match is found, the position of the match is returned. If no match is found after checking all possible positions, a message indicating no match is printed.

**Algorithm:**

* 1. Start
  2. Read two input strings, T (text) and P (pattern).
  3. Compute the lengths of the text (S) and the pattern (R).
  4. Iterate through each possible starting index in the text (K), from 0 to S - R.
  5. For each starting index K, check if the substring starting at K matches the pattern by comparing each character.
  6. If a match is found, print the 1-based index of the match position and stop the search.
  7. If no match is found after checking all positions, print "No match found."
  8. End.

**Source Code:**

#include <iostream>

using namespace std;

int main() {

    string T, P;

       cin >> T >> P;

    cout << "Text (T): " << T << endl;

    cout << "Pattern (P): " << P << endl;

    int R = P.size(), S = T.size();

    for (int K = 0; K <= S - R; K++) {

        bool match = true;

        for (int L = 0; L < R; L++) {

            if ( P[L] != T[K + L] ) {

                match = false;

                break;

            }

        }

        if (match) {

            cout << "Match found at position: " << K + 1 << endl;

            return 0;

        }

    }

    cout << "No match found" << endl;

    return 0;

}

**Input:**

Text(T): aabbbabc

Pattern(P): abc

**Output:**

Text(T): aabbbabc

Pattern(P): abc

Pattern found at position: 5

**Experiment No:** 06

**Experiment Name**: **Implement a queue data structure along with its typical operations.**

**Theory:**

A queue is a linear data structure that follows the FIFO (First In, First Out) principle. In a queue, elements are added from the rear (enqueue) and removed from the front (dequeue). The operations associated with a queue are:

1. Enqueue: Add an element to the rear of the queue.
2. Dequeue: Remove an element from the front of the queue.
3. Display: Display the elements of the queue.

**Algorithm:**

 Start

 If Operation is "Enqueue":

* If rear == SIZE - 1, display "Queue Overflow!" and terminate.
* If front == -1, set front = 0 (initializing the queue).
* Increment rear by 1.
* Set queue[rear] = value (insert value at the rear).
* Display "value has been added to the queue."

 Else if Operation is "Dequeue":

* If front == -1, display "Queue Underflow!" and terminate.
* If front == rear, display "queue[front] has been removed" and set both front = -1 and rear = -1 (reset the queue).
* Else, display "queue[front] has been removed" and increment front by 1 (remove the front element).

 End

**Source Code:**

#include <iostream>

using namespace std;

#define SIZE 5

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

// Enqueue operation

void enqueue(int value) {

    if (rear == SIZE - 1) {

        cout << "Queue Overflow! (Queue is full)\n";

    } else {

        if (front == -1) front = 0;

        rear++;

        queue[rear] = value;

        cout << value << " has been added.\n";

    }

}

// Dequeue operation (Removing data)

void dequeue() {

    if (front == -1 && rear==-1) {

        cout << "Queue Underflow! (Queue is empty)\n";

    } else if (front==rear){

        cout << queue[front] << " has been removed.\n";

        front = rear = -1;

    }

    else{

        cout<<queue[front]<<" has been removed.\n";

        front++;

    }

}

void display() {

    if (front == -1) {

        cout << "Queue is empty!\n";

    } else {

        cout << "Queue: ";

        for (int i = front; i <= rear; i++) {

            cout << queue[i] << " ";

        }

        cout << endl;

    }

}

// Main function

int main() {

    enqueue(10);

    enqueue(20);

    enqueue(30);

    display();

    dequeue();

    display();

    enqueue(40);

    enqueue(50);

    display();

    dequeue();

    display();

    return 0;

}

**Output:**

10 has been added.

20 has been added.

30 has been added.

**Queue: 10 20 30**

10 has been removed.

**Queue: 20 30**

40 has been added.

50 has been added.

**Queue: 20 30 40 50**

20 has been removed.

**Queue: 30 40 50**

**Experiment No: 07**

**Experiment Name : Solving N-Queens Problem Efficiently Using Backtracking Algorithm.**

**Theory :**

Backtracking is an algorithmic technique for solving optimization problems where solutions are built incrementally. It systematically explores all possible configurations and rejects those that are not feasible, thus "backtracking" when it reaches an invalid state.

For the N-Queens problem, backtracking is used to explore all valid ways to place queens on the chessboard. The algorithm attempts to place queens one by one in each row and checks whether the position is safe. If a solution is found, it is printed. If no solution is possible in a particular configuration, the algorithm backtracks and tries a different position.

**Algorithm :**

 **Start with the first row**: The first queen is placed in the first row at the first column.

 **Place queens row by row**: For each row, attempt to place a queen in each column.

 **Check for safety**: For each column position, check whether placing the queen would result in a conflict with already-placed queens. The conflict check includes:

* Column conflict: No two queens should share the same column.
* Diagonal conflict: No two queens should lie on the same diagonal.

 **Backtrack if necessary**: If placing a queen results in a conflict, move the queen to the next possible column. If no valid position is found, backtrack to the previous row and try a new column for the queen placed there.

 **Base case**: If all queens are placed successfully, print the board and backtrack to find other possible solutions.

 **Repeat** until all solutions are found.

**Source Code :**

#include <iostream>

#include <cmath>

using namespace std;

int board[20];

int count = 0;

// Function to print the board

void print\_board(int n) {

    cout << "\nSolution " << ++count << ": \n";

    for (int i = 1; i <= n; i++) {

        for (int j = 1; j <= n; j++) {

            if (board[i] == j)

                cout << "\tQ";  // Queen's position

            else

                cout << "\t-";  // Empty slot

        }

        cout << endl;

    }

}

// Function to check if placing a queen is safe

bool place(int row, int column) {

    for (int i = 1; i < row; i++) {

        if (board[i] == column || abs(board[i] - column) == abs(i - row)) {

            return false;

        }

    }

    return true;

}

// Function to solve n-Queens using backtracking

void Queen(int row, int n) {

    for (int column = 1; column <= n; column++) {

        if (place(row, column)) {

            board[row] = column;

            if (row == n)

                print\_board(n);

            else

                Queen(row + 1, n);

        }

    }

}

int main() {

    int n;

    cout << "Enter number of Queens: ";

    cin >> n;

    count = 0;

    Queen(1, n);

    cout << "\nTotal number of solutions: " << count << endl;  // Display total solutions

    return 0;

}

**Input :**

**Enter number of queens : 4**

**Output :**

**Solution 1:**

**- Q - -**

**- - - Q**

**Q - - -**

**- - Q -**

**Solution 2:**

**- - Q -**

**Q - - -**

**- - - Q**

**- Q - -**

**Total number of solutions: 2**

**Experiment No: 08**

**Experiment Name: Subset Sum Problem Using Backtracking**

**Theory:**

The Subset Sum Problem is a classic problem in computer science, where you are given a set of integers, and the task is to determine whether there is a subset of these integers that adds up to a specific target sum d. The problem can be solved using several techniques, and one of the most effective methods is backtracking.

Backtracking involves incrementally building solutions and abandoning those that fail to meet the criteria. It is an exhaustive search that systematically checks all possible subsets of the set.

Let the set SSS be defined as:

S={5,10,12,13,15,18}

The target sum d=is the value we want to check for, to find if there exists any subset of S that sums up to d.

**Algorithm:**

 **Step 1:** Start with an empty subset.

 **Step 2:** Add the next element from the set to the subset.

 **Step 3:** If the sum of the subset is equal to the target sum d, stop and print the subset as the solution.

 **Step 4:** If the sum exceeds the target or if we reach the end of the set, backtrack by removing the last added element.

 **Step 5:** If the subset is valid, repeat step 2.

 **Step 6:** If no solution is found after exploring all possibilities, stop without a solution.

**Source Code:**

#include <iostream>

#include <vector>

using namespace std;

// Function to print the current subset

void printSubset(const vector<int>& subset) {

    cout << "{ ";

    for (int i = 0; i < subset.size(); i++) {

        cout << subset[i] << " ";

    }

    cout << "}" << endl;

}

// Function to find subsets

void findSubsetSum(const vector<int>& S, int d, vector<int>& subset, int currentSum, int index) {

    if (currentSum == d) {

        printSubset(subset);

        return;

    }

       if (index == S.size() || currentSum > d) {

        return;

    }

       subset.push\_back(S[index]);

    findSubsetSum(S, d, subset, currentSum + S[index], index + 1);

    subset.pop\_back();

    findSubsetSum(S, d, subset, currentSum, index + 1);

}

int main() {

       vector<int> S = {5, 10, 12, 13, 15, 18};

    int d = 30;

    vector<int> subset;

    cout << "Subsets with sum " << d << " are:\n";

    findSubsetSum(S, d, subset, 0, 0);  // Call the function to find subsets

    return 0;

}

**Input:**

 Set S = {5, 10, 12, 13, 15, 18}

 Target sum d = 30

**Output:**

**Subsets with sum 30 are:**

**{** 5 10 15 **}**

**{ 5 12 13 }**

**{ 12 18 }**

**Experiment No: 09**

**Experiment Name: Solving the 0/1 Knapsack Problem using Dynamic Programming**

**Theory:**

The **0/1 Knapsack Problem** is a combinatorial optimization problem where we are given a set of items, each with a profit and a weight. The goal is to select a subset of items that maximizes the total profit without exceeding a given weight capacity. The term "0/1" refers to the fact that each item is either included (1) or excluded (0) from the knapsack.

**Problem Formulation:**

* We have n items, each with a profit pip\_ipi​ and weight wiw\_iwi​.
* A knapsack with a maximum weight capacity W.
* We aim to maximize the total profit ∑pi\sum p\_i∑pi​ while ensuring ∑wi≤W\sum w\_i \leq W∑wi​≤W.

**Dynamic Programming Approach:**

To solve this problem, we use dynamic programming (DP) to compute the maximum profit in a 2D table dp[i][w] where:

* i represents the first i items considered.
* w represents the knapsack's capacity.

The DP recurrence relation is:

dp[i][w]=max⁡(dp[i−1][w],pi+dp[i−1][w−wi])dp[i][w] = \max(dp[i-1][w], p\_i + dp[i-1][w - w\_i])dp[i][w]=max(dp[i−1][w],pi​+dp[i−1][w−wi​])

This equation states that for each item, we either exclude it (take the value from dp[i-1][w]) or include it (add its profit and check the remaining capacity).

**Algorithm:**

1. Start
2. Input the number of items n and the knapsack capacity W.
3. Input the profits and weights of each item.
4. Initialize a 2D array dp[0..n][0..W] where dp[i][w] stores the maximum profit achievable with the first i items and a knapsack capacity w.
5. **Fill the DP table**:
   * For each item i from 1 to n, and for each capacity w from 0 to W:
     + If item i can be included (i.e., w\_i <= w), compute the maximum of:
       - **Not including the item**: dp[i-1][w]
       - **Including the item**: profits[i-1] + dp[i-1][w - weights[i-1]]
     + Otherwise, set dp[i][w] = dp[i-1][w] (exclude the item).
6. **Backtrack** through the DP table to find the items included in the optimal solution:
   * If dp[i][w] != dp[i-1][w], then item i is included in the knapsack, and reduce the capacity w = w - weights[i-1].
7. **Output** the maximum profit and the list of selected items.
8. **End**

**Source Code:**

#include <iostream>

using namespace std;

int main() {

    int profits[100], weights[100];

    int dp[101][101];

    int n, capacity;

    cout << "Enter the number of items: ";

    cin >> n;

    cout << "Enter the profits of the items:\n";

    for (int i = 0; i < n; i++) {

        cin >> profits[i];

    }

    cout << "Enter the weights of the items:\n";

    for (int i = 0; i < n; i++) {

        cin >> weights[i];

    }

    cout << "Enter the capacity of the knapsack: ";

    cin >> capacity;

    // DP Table initialization

    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] = (profits[i - 1] + dp[i - 1][w - weights[i - 1]] > dp[i - 1][w]) ?

                            profits[i - 1] + dp[i - 1][w - weights[i - 1]] : dp[i - 1][w];

            } else {

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

            }

        }

    }

    cout << "DP Table:\n";

    for (int i = 0; i <= n; i++) {

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

            cout << dp[i][w] << " ";

        }

        cout << endl;

    }

    int w = capacity;

    cout << "\nItems included to achieve maximum profit:\n";

    for (int i = n; i > 0; i--) {

        if (dp[i][w] != dp[i - 1][w]) {

            cout << "Item " << i << " (Profit: " << profits[i - 1] << ", Weight: " << weights[i - 1]  << ")\n";

            w = w - weights[i - 1];

        }

    }

    cout << "Maximum profit in the knapsack: " << dp[n][capacity] << endl;

    return 0;

}

**Input:**

Enter the number of items: 4

Enter the profits of the items: 15 25 13 23

Enter the weights of the items: 2 6 12 9

Enter the capacity of the knapsack: 20

**Output:**

**DP Table:**

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15

0 0 15 15 15 15 25 25 40 40 40 40 40 40 40 40 40 40 40 40 40

0 0 15 15 15 15 25 25 40 40 40 40 40 40 40 40 40 40 40 40 53

0 0 15 15 15 15 25 25 40 40 40 40 40 40 40 48 48 63 63 63 63

Items included to achieve maximum profit:

Item 4 (Profit: 23, Weight: 9)

Item 2 (Profit: 25, Weight: 6)

Item 1 (Profit: 15, Weight: 2)

Maximum profit in the knapsack: 63

**Experiment No: 10**

**Experiment Name**: **Solving the Tower of Hanoi Problem using an Iterative Approach with a Stack.**

**Theory:**

The **Tower of Hanoi** is a classic puzzle that involves three pegs and a number of disks of different sizes. The goal is to move all the disks from the source peg to the destination peg following these three rules:

1. Only one disk can be moved at a time.
2. A disk can only be placed on top of a larger disk or on an empty peg.
3. A disk can only be moved to another peg.

The problem can be solved recursively with the minimum number of moves being 2^n - 1, where n is the number of disks. This solution, however, can be implemented iteratively using a stack to simulate recursion. This iterative approach is helpful when recursion may cause issues, such as stack overflow with a very large number of disks.

**Algorithm:**

 **Start**

 **Initialize stack** with the initial state:

* n (number of disks),
* source,
* destination,
* auxiliary

 **While stack is not empty**:

* **Pop** a state from the stack.
* **If** n == 1:
  + Print the move: source -> destination.
* **Else**:
  + Push subproblems onto the stack:
    1. Move n-1 disks from auxiliary to destination.
    2. Move 1 disk from source to destination.
    3. Move n-1 disks from source to auxiliary.

 **End** when the stack is empty.

**Source Code:**

#include <iostream>

using namespace std;

int main() {

    int num;

    char source, destination, auxiliary;

    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;

    cout << "Total number of moves: " << total\_moves << endl;

    int stack[1000][4];

    int top = -1;

    // Push initial state onto the stack

    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;

            // Move 1 disk from source to destination

            stack[++top][0] = 1;

            stack[top][1] = from\_peg;

            stack[top][2] = to\_peg;

            stack[top][3] = aux\_peg;

            // Move n-1 disks from source to auxiliary

            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 -> B**

**A -> C**

**B -> C**

**A -> B**

**C -> A**

**C -> B**

**A -> B**