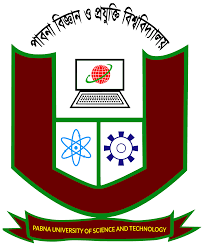
**Pabna University of Science and Technology**



**Faculty of Engineering and Technology**

**Department of Information and Communication Engineering**

**Lab Report**

Course Code: **ICE-2202**

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

|  |  |
| --- | --- |
| **Submitted By:**  Name: Mst. Khaleda Khatun  Roll: 220640  Session:2021-2022  2nd Year 2ndsemester  Department of Information and  Communication Engineering,  PUST. | **Submitted To:**  **Name: MD. Anwar Hossain**  Chairman and Professor,  Department of Information and Communication Engineering,  Pabna University of Science And Technology,Pabna. |

**Date of Submission: 01/03/2025**

**Laboratory Problem Index**

**Course:** ICE-2202 - Data Structure and Algorithm Sessional  
**Institution:**Pabna University of Science and Technology  
**Session:** 2021-2022

**Index of Laboratory Problems**

| **Sl. No** | **Problem Description** |
| --- | --- |
| **1** | Bubble Sort - Sorting a linear array |
| **2** | Linear Search - Finding an element in an array |
| **3** | Merge Sort - Sorting a linear array |
| **4** | Binary Search - Finding an element in an array |
| **5** | Pattern Matching - Finding a given pattern in text |
| **6** | Queue Implementation - Basic operations of a queue |
| **7** | N-Queens Problem - Using backtracking |
| **8** | Subset Sum Problem - Given a set S and sum d |
| **9** | 0/1 Knapsack Problem - Dynamic programming approach |
| **10** | Tower of Hanoi - Solving for N disks |

**No of Experiment**: 01

**Name of Experiment:** A program to sort a linear array using the bubble sort algorithm.

**Objective:**

* To understand and implement the **Bubble Sort** algorithm.
* To analyze its **step-by-step execution** and **time complexity**.
* To sort a linear array by comparing and switching its elements

**Theory:**

Bubble Sort is a straightforward comparison-based sorting algorithm that repeatedly goes through the array, swapping adjacent elements when they are out of order. This process keeps going until the whole array is sorted.

1. It is a stable sorting algorithm.

2. It performs well with small arrays but becomes inefficient with larger datasets.

3. It can be optimized by halting early if the array gets sorted before all passes are finished.

**Algorithm:**

1. Start by comparing adjacent elements in the array.
2. If the left element is greater than the right, swap them.
3. Move to the next pair and repeat the process for the entire array.
4. After the first pass, the largest element is placed at the last index.
5. Repeat the process for the remaining unsorted elements.
6. If no swaps occur in a pass, terminate the algorithm early (optimization).
7. The array is sorted when no further swaps are needed.

**Source Code:**

#include <iostream>

using namespace std;

int main() {

int n;

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

cin>> n;

int arr[n];

cout<< "Enter " << n << " elements: ";

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

cin>>arr[i];

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

cout<< "\nPass " <<i + 1 << ":\n";

bool swapped = false;

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

if (arr[j] >arr[j + 1]) {

cout<< "Swapping " <<arr[j] << " and " <<arr[j + 1] <<endl;

swap(arr[j], arr[j + 1]);

swapped = true;

}

}

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

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

cout<<arr[k] << " ";

}

cout<<endl;

if (!swapped) break; // Stop if already sorted

}

cout<< "\nFinal sorted array: ";

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

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

}

return 0;

}

**Output:**

Enter the size of the array: 5

Enter 5 elements: 5 3 8 4 2

**Pass 1:**

Swapping 5 and 3

Swapping 8 and 4

Swapping 5 and 2

Array after pass 1: 3 5 4 2 8

**Pass 2:**

Swapping 5 and 4

Swapping 5 and 2

Array after pass 2: 3 4 2 5 8

**Pass 3:**

Swapping 4 and 2

Array after pass 3: 3 2 4 5 8

**Pass 4:**

Swapping 3 and 2

Array after pass 4: 2 3 4 5 8

**Final sorted array: 2 3 4 5 8**

**No of Experiment:** 02

**Name of Experiment:** A program to find an Element using a Linear search algorithm.

**Objective:**

* To understand and apply the **Linear Search algorithm**.
* To locate an element in a specified array by examining each element sequentially.
* To evaluate the **time complexity** of the Linear Search approach.

**Theory:**

Linear Search is a  **basic and easy to understand searching algorithm** that sequentially checks **all element of an array** until the desired element is found .This method works best for small arrays or datasets that are not sorted.

**Algorithm:**

 Start from the first element of the array.

 Compare each element with the search key.

 If a match is found, return the index of the element.

 If no match is found after checking all elements, return -1.

**Source Code:**

#include <iostream>

using namespace std;

int main() {

int n, key;

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

cin>> n;

intarr[n];

cout<< "Enter " << n << " elements: ";

for (inti = 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 found at index " <<i<<endl;

return 0; }

}

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

return 0;}

**Output:**

Enter the size of the array: 7

Enter 5 elements: 11 22 33 44 55 66 77

Enter the element to search: 77

Element found at index 6

**No of Experiment:** 03

**Name of Experiment:** A program to sort a linear array using the marge sort algorithm.

**Objective:**

 Understand how the Merge Sort algorithm works.

 Apply Merge Sort using C++.

 Solve its time complexity and efficiency.

**Theory:**

Merge Sort is a **sharing and conquer** algorithm that recursively divides an array into two divides , sorts each half, and then merges them back together in sorted order. It combine the sorted halfs into a single sorted array.

**Algorithm:**

1. If the array has one or no elements, return (base case).

2. Find the middle index of the array.

3. Recursively apply Merge Sort on the left half of the array.

4. Recursively apply Merge Sort on the right half of the array.

5. Merge the two sorted halves:

-Compare elements from both halves.

-Copy the smaller element into the main array.

- Copy any remaining elements from both halves.

6. Return the sorted array.

**Source Code:**

#include <iostream>

using namespace std;

void merge(int arr[], int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

int L[n1], R[n2];

for (int i = 0; i < n1; i++)

L[i] = arr[left + i];

for (int j = 0; j < n2; j++)

R[j] = arr[mid + 1 + j];

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

void mergeSort(int arr[], int left, int right) {

if (left < right) {

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

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

void printArray(int arr[], int size) {

for (int i = 0; i < size; i++)

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

cout << endl;

}

int main() {

int arr[] = {38, 27, 43, 3, 9, 82, 10};

int size = sizeof(arr) / sizeof(arr[0]);

cout << "Original array: ";

printArray(arr, size);

mergeSort(arr, 0, size - 1);

cout << "Sorted array: ";

printArray(arr, size);

return 0;

}

**Output:**

Original array: 38,27,43,3,9,82,10

Sorted array: 3,9,10,27,38,43,82

**No of Experiment**: 04

**Name of Experiment:** A program to find an element using the binary search algorithm.

**Objective:**

* To understand the concept of searching using the Binary Search algorithm.
* To implement the Binary Search algorithm in C++.
* To study the time complexity of Binary Search.

**Theory:**

Binary Search is an **effective searching algorithm** that works on **a sorted array**. If is an extremely efficient searching algorithm that can be sorted DATAin increasing numerical order or equivalent, alphabetical. We follow this:

1. Find the **middle element** of the array.
2. If the middle element is equal to the key, return its index.
3. If the key is **less than** the middle element, search in the **left half**.
4. If the key is **greater than** the middle element, search in the **right half**.
5. Repeat the process until the element is found or the array is fully searched.

**Algorithm:**

1. Set low = 0 and high = n - 1.
2. Repeat while low <= high:
   * Find mid = (low + high) / 2.
   * If arr[mid] == key, return mid.
   * If arr[mid] > key, set high = mid - 1.
   * Else, set low = mid + 1.

3.If the element is not found, return -1.

**Source Code:**

#include <iostream>

using namespace std;

int main() {

int n, key;

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

cin>> n;

intarr[n];

cout<< "Enter the sorted elements: ";

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

cin>>arr[i];

}

cout<< "Enter the element to search: ";

cin>> key;

int low = 0, high = n - 1, mid, index = -1;

while (low <= high) {

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

if (arr[mid] == key) {

index = mid;

break;

} else if (arr[mid] > key) {

high = mid - 1;

} else {

low = mid + 1;

}

}

if (index != -1)

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

else

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

return 0;

}

**Output:**

Enter the number of elements: 7  
Enter the sorted elements: 15 20 35 40 50  
Enter the element to search: 40  
Element found at index: 3

**No of Experiment: 05**

**Name of Experiment : A program to find a given pattern from text using the pattern matching algorithm.**

**Theory:**

Pattern matching is a crucial concept in computer science and is widely used in applications such as text processing, search engines, and DNA sequencing. The goal of pattern matching is to search event of a given pattern PPP in a larger text TTT.

There are several pattern matching algorithms, including:

1. Brute Force Algorithm - Compares the pattern with the text sequentially.
2. Knuth-Morris-Pratt (KMP) Algorithm - Uses preprocessing to avoid additional comparisons.
3. Rabin-Karp Algorithm - Uses hashing for efficient search.
4. Boyer-Moore Algorithm - Skips unnecessary comparisons using heuristics.

In this lab, we focus on the Knuth-Morris-Pratt (KMP) Algorithm, which preprocesses the pattern using a Longest Prefix Suffix (LPS) array to optimize the searching process.

**Objective:**

* To realize the concept of pattern matching in data structures and algorithms.
* To implement the KMP Algorithm for finding a pattern within a text.
* To analyze the efficiency of the algorithm in terms of time complexity.

### Algorithm:

**Time Complexity:**

• Naive Algorithm: O(n \* m), where n is the text size and m is the pattern size.

• KMP Algorithm: O(n + m).

• Boyer-Moore Algorithm: O(n/m) in the worst case.

Algorithm (Naive Approach):

1. Take the text and pattern as input.

2. Retrieve the length of the text (n) and pattern (m).

3. Loop through the text from index i = 0 to i = n-m.

4. For each position i, match the pattern with the substring of the text.

5. If all the characters are matched, then print the position where the pattern is encountered.

6. If there is no match, print "Pattern not found".

**Source Code:**

#include <iostream>

#include <string>

using namespace std;

int main() {

string text, pattern;

cout<< "Enter the text: ";

getline(cin, text);

cout<< "Enter the pattern to search: ";

getline(cin, pattern);

int n = text.length(), m = pattern.length();

bool found = false;

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

int j;

for (j = 0; j < m; j++) {

if (text[i + j] != pattern[j])

break;

}

if (j == m) {

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

found = true;

}

}

if (!found)

cout<< "Pattern not found" <<endl;

return 0;

}

**Output:**

Enter the text: Creator   
Enter the pattern to search: Tor  
Pattern found at index 5

**No of Experiment: 06**

**Name of Experiment:** A program to implement a queue data structure along with its typical operation

**Objective:**

* To understand the concept of the **Queue** data structure.
* To implement a **Queue** using an **array** in C++.
* To perform basic queue operations: **Enqueue (Insertion), Dequeue (Deletion), Peek (Front element), and Display (Traversal).**
* To analyze the time complexity of queue operations.

**Theory:**

A **queue** is a **linear data structure** that follows the **FIFO (First-In, First-Out) principle. It** means that the element inserted first will be removed first.

### **Some Operations on Queue:**

* **Enqueue:** Adds an element to the rear of the queue.
* **Dequeue:** Removes an element from the front of the queue.
* **Peek (Front):** Returns the front element without removing it.
* **IsEmpty:** Checks if the queue is empty.
* **IsFull:** Checks if the queue is full for array implementation.

### **Some applications of Queue**:****

* **Job scheduling** in operating systems.
* **Printer task scheduling.**
* **Handling requests in web servers.**

**Algorithm:**

1. **Initialize the Queue**

* Set front = -1, rear = -1.

2.**Enqueue (Insert an element)**

* If rear == SIZE - 1, print **"Queue is full"**.
* Else:
  + If front == -1, set front = 0.
  + Increase rear by 1.
  + Insert the element at arr[rear].

3. **Dequeue (Remove an element)**

* If front == -1 or front > rear, print **"Queue is empty"**.
* Else:
  + Print and remove the front element.
  + Increase front by 1.

4**.Display (Print all elements)**

* If queue is empty, print **"Queue is empty"**.
* Else, print all elements from front to rear.

5. **Exit the Program**

* Stop when the user chooses exit.

**Source Code:**

#include <iostream>

using namespace std;

#define SIZE 5

intarr[SIZE];

int front = -1, rear = -1;

int main() {

int choice, value;

while (true) {

cout<< "\nQueue Operations:\n";

cout<< "1. Enqueue\n2. Dequeue\n3. Display\n4. Exit\n";

cout<< "Enter your choice: ";

cin>> choice;

if (choice == 1) {

if (rear == SIZE - 1) {

cout<< "Queue Overflow!" <<endl;

continue;

}

cout<< "Enter value to enqueue: ";

cin>> value;

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

arr[++rear] = value;

cout<< "Inserted " << value << " into the queue." <<endl;

}

else if (choice == 2) {

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

cout<< "Queue Underflow!" <<endl;

continue;

}

cout<< "Removed " <<arr[front++] << " from the queue." <<endl;

}

else if (choice == 3) {

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

cout<< "Queue is empty!" <<endl;

continue;

}

cout<< "Queue elements: ";

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

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

}

cout<<endl;

}

else if (choice == 4) {

break;

}

else {

cout<< "Invalid choice!" <<endl;

}

}

return 0;

}

**Output:**

1. Enqueue 10

Inserted 10 into the queue.

2. Enqueue 20

Inserted 20 into the queue.

3. Enqueue 30

Inserted 30 into the queue.

1. Display

Queue elements: 10 20 30

2. Dequeue

Removed 10 from the queue.

3. Display

Queue elements: 20 30

4. Exit

**No of Experoment:** 07

**Name of Experoment:**A program to solve n queen’s problem using backtracking.

**Objective:**

* To realize the N-Queens Problem and implement it using backtracking.
* To place N queens on an N × N chessboard such that no two queens attack each other.
* To explore the recursive approach in solving constraint satisfaction problems.

**Theory:**

The **N-Queens Problem** is a classic combinatorial problem where **N queens** must be placed on an **N × N chessboard** so that:

1. No two queens are in the **same row**.
2. No two queens are in the **same column**.
3. No two queens are on the **same diagonal**.

### Backtracking is a **trial-and-error** approach where we:

1. Place a queen in a row.
2. Check if it leads to a valid configuration.
3. If valid, proceed to the next row.
4. If invalid, **backtrack** and try another position.
5. Repeat until all queens are placed.

### **Some applications of N-Queens Problem**

* AI and **constraint satisfaction problems**
* **Robotics pathfinding**
* **Genetic algorithms** and **optimization problems**

**Algorythm:**

### **Initialize the Board**

* Use a **1D array board[N]** where board[i] stores the **row position** of the queen in column i.
* Example: If board[2] = 3, it means a queen is placed in **column 2, row 3**.

### **Check if a Position is Safe (**isSafe **function**)****

* A queen can **attack horizontally or diagonally**.
* For every previous column i, check:
  + If the same row is occupied (board[i] == row).
  + If it is diagonally attacked (abs(board[i] - row) == abs(i - col)).

### **Solve the Problem Using Backtracking (**solve **function)**

* If all columns are filled (col >= N), return true (solution found).
* Try placing a queen in each row of the current column:
  + If isSafe(row, col), place a queen at board[col] = row.
  + **Recursively call solve(col + 1)** to place queens in the next column.
  + If placing the queen leads to a solution, return true.
  + If not, backtrack and try the next row.

### **Print the Solution (**printBoard **function)**

* Loop through the board and print 1 where the queen is placed and 0 elsewhere.

**Source Code:**

#include <iostream>

using namespace std;

#define N 8

int board[N] = {0};

bool isSafe(int row, int col) {

for (int i = 0; i < col; i++)

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

return false;

return true;

}

bool solve(int col) {

if (col >= N) return true;

for (int row = 0; row < N; row++) {

if (isSafe(row, col)) {

board[col] = row;

if (solve(col + 1)) return true;

}

}

return false;

}

void printBoard() {

for (int i = 0; i < N; i++, cout << endl)

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

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

}

int main() {

if (solve(0)) printBoard();

else cout << "No solution";

}

**Output:**

[1] 0 0 0 0 0 0 0

0 0 0 0 [1] 0 0 0

0 0 0 0 0 0 0 [1]

0 0 0 0 0 [1] 0 0

0 0 [1] 0 0 0 0 0

0 0 0 0 0 0 [1] 0

0 [1] 0 0 0 0 0 0

0 0 0 [1] 0 0 0 0

All possible solutions displayed above!

**No of Experoment:** 08

**Name of Experoment:** Consider a set S ={5,10,12,13,15,18} and d=30.Write a program to solve sum of subset problem.

**Objective:**

1.To comprehend and apply backtracking for solving the subset sum problem.

2.To effectively produce only the valid subsets.

3.To evaluate the time complexity and optimization strategies utilized in backtracking

**Theory:**

The Sum of Subset problem is a well-known challenge in combinatorial optimization and backtracking. It focuses on identifying all possible subsets of a given set S that add up to a specific value d. For instance, with the set S = {5,10,12,13,15,18} and a target sum d = 30, our goal is to find all subsets that total 30. This problem can be approached using backtracking, which effectively navigates through subsets while eliminating unnecessary recursive calls. The method ensures that we only generate the subsets that lead to the desired sum, rather than all possible combinations.

**Algorithm:**

### **Step 1: Input the Set and Target Sum**

* Define the set **S** and the required sum **d.**

### **Step 2: Recursive Function (Backtracking)**

1. Start with an empty subset and sum = 0.
2. Explore each element recursively:
   * Include the current element in the subset and proceed.
   * Exclude the current element and proceed.
3. If at any stage sum equals **d,** print the subset.
4. If sum exceeds **d**, backtrack.
5. Continue exploring until all possible subsets are checked.

### **Step 3: Display Results**

* Print all valid subsets.

**Source Code:**

#include <iostream>

#include <vector>

using namespace std;

void findSubsets(vector<int> &arr, vector<int> &subset, int index, int sum, int target) {

if (sum == target) {

cout << "Subset found: ";

for (int num : subset) cout << num << " ";

cout << endl;

return;

}

if (index == arr.size() || sum > target) return;

subset.push\_back(arr[index]);

findSubsets(arr, subset, index + 1, sum + arr[index], target);

subset.pop\_back();

findSubsets(arr, subset, index + 1, sum, target);

}

int main() {

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

int target = 30;

vector<int> subset;

cout << "Finding subsets that sum to " << target << "...\n";

findSubsets(S, subset, 0, 0, target);

return 0;

}

**Output:**

Finding subsets that sum to 30...

Subset found: 5 10 15

Subset found: 5 12 13

Subset found: 12 18

**No of Experiment: 09**

**Name of Experiment: Write a program to solve the following 0/1 Knapsack using dynamic programming approach profit P=(15,25,13,23) ,weight w =(2,6,12,9), Knapsack C= 20, and the number of item n=4.**

**Objective:**

* To understand and implement the 0/1 Knapsack problem using the DynamicProgramming approach.
* To maximize the total profit without exceeding the capacity constraint of the knapsack.
* To analyze the time complexity of the Dynamic Programming solution.

**Theory:**

The **0/1 Knapsack problem** is a classic optimization problem where:

* We have n items, each with a **profit (P[i])** and **weight (W[i]).**
* We need to select a subset of these items to **maximize the total profit** while ensuring that the total weight does not exceed the **capacity (C)** of the knapsack.
* **0/1 means** that an item can either be **fully included (1) or not included at all (0)** (no fractional selection).

### **Dynamic Programming Approach**:****

* We use a **2D table** dp[i][j] where:
  + i represents the first i items.
  + j represents the knapsack capacity from 0 to C.
* **Recurrence Relation:** dp[i][j]={dp[i−1][j]if W[i−1]>j (Item cannot be included)max⁡(dp[i−1][j],P[i−1]+dp[i−1][j−W[i−1]])otherwise (Include or Exclude item)dp[i][j] = \begin{cases} dp[i-1][j] & \text{if } W[i-1] > j \text{ (Item cannot be included)} \\ \max(dp[i-1][j], P[i-1] + dp[i-1][j - W[i-1]]) & \text{otherwise (Include or Exclude item)} \end{cases}dp[i][j]={dp[i−1][j]max(dp[i−1][j],P[i−1]+dp[i−1][j−W[i−1]])​if W[i−1]>j (Item cannot be included)otherwise (Include or Exclude item)​
* **Base Case:**
  + dp[0][j] = 0 (If no items are available, the maximum profit is 0).
  + dp[i][0] = 0 (If knapsack capacity is 0, the maximum profit is 0).

**Algorithm:**

1. Create a 2D DP table of size (n+1) × (C+1).
2. Initialize base cases (dp[0][j] = 0 and dp[i][0] = 0).
3. For each item i = 1 to n:
   * For each weight j = 1 to C:
     + If W[i-1] > j:
       - dp[i][j] = dp[i-1][j] (Exclude the item)
     + Else:
       - dp[i][j] = max(dp[i-1][j], P[i-1] + dp[i-1][j - W[i-1]]) (Include or Exclude)
4. Return dp[n][C] as the maximum profit.

**Source Code:**

#include <iostream>

#include <vector>

using namespace std;

// Function to solve 0/1 Knapsack using Dynamic Programming

int knapsack(int P[], int W[], int C, int n) {

vector<vector<int>> dp(n + 1, vector<int>(C + 1, 0)); // DP table

// Build the DP table

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

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

if (W[i - 1] > j) {

dp[i][j] = dp[i - 1][j]; // Item cannot be included

} else {

dp[i][j] = max(dp[i - 1][j], P[i - 1] + dp[i - 1][j - W[i - 1]]);

}

}

}

// Print the DP table

cout << "\nDP Table:\n";

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

for (int j = 0; j <= C; j++) {

cout << dp[i][j] << "\t";

}

cout << endl;

}

// Maximum profit is stored at dp[n][C]

return dp[n][C];

}

int main() {

int P[] = {15, 25, 13, 23}; // Profits

int W[] = {2, 6, 12, 9}; // Weights

int C = 20; // Knapsack capacity

int n = sizeof(P) / sizeof(P[0]); // Number of items

cout << "Maximum Profit: " << knapsack(P, W, C, n) << endl;

return 0;

}

**Input:**

Profits: {15, 25, 13, 23}

Weights: {2, 6, 12, 9}

Knapsack Capacity: 20

**Output:**

**Maximum Profit: 53**

**No of Experiment: 10**

**Name of Experiment:** Write a program to solve the tower of Hanoi problem from the N disk

**Objective:**

* To realize the concept of recursion.
* To prove the Tower of Hanoi algorithm in C++.
* To realize the recursive nature of the solution.

**Theory:**

The **Tower of Hanoi** is a mathematical puzzle that involves to three rods and **N** disks of different sizes. The objective is to move all the 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 top of a smaller disk.
3. Only the top disk of a rod can be moved.

### Algorithm:

### **Step 1: Define the problem**

* There are **N** disks and three rods**: Source (A), Auxiliary (B), and Destination (C).**

### **Step 2: Recursive Approach**

1. Move **N-1** disks from **Source (A)** to **Auxiliary (B)** using **Destination (C).**
2. Move the **Nth (largest) disk** from **Source (A)** to **Destination (C)**.
3. Move the **N-1** disks from **Auxiliary (B)** to **Destination (C)** using **Source (A)**.

### **Step 3: Base Condition**

* If there is only one disk, move it directly from **Source (A) to Destination (C).**

**Source Code:**

#include <iostream>

using namespace std;

void towerOfHanoi(int n, char source, char auxiliary, char destination) {

if (n == 1) {

cout << "Move disk 1 from " << source << " to " << destination << endl;

return;

}

towerOfHanoi(n - 1, source, destination, auxiliary);

cout << "Move disk " << n << " from " << source << " to " << destination << endl;

towerOfHanoi(n - 1, auxiliary, source, destination);

}

int main() {

int n;

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

cin >> n;

towerOfHanoi(n, 'A', 'B', 'C');

return 0;

}

**Output:**

Enter the number of disks: 3

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