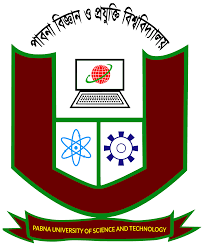
**Pabna University of Science and Technology**



**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:** Rahat Mahbub  **Roll:** 220641  **Reg:** 1065488  **Session:**2021-2022  2 nd Year 2 nd semester  Department of Information and  Communication Engineering,  PUST | **Submitted To:**  **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 |

**Experiment No:** 1

**Name of the Experiment:**

A program to sort a linear array using the Bubble Sort algorithm.

**Objectives:**

1. To understand the concept of sorting using the Bubble Sort algorithm.

2. To implement the Bubble Sort algorithm in C++.

3. To analyze the time complexity of Bubble Sort.

**Theory:**

Bubble Sort is a simple sorting algorithm that repeatedly traverses the list, comparing and swapping adjacent items if they are in the wrong order. This is continued until the list is sorted. The algorithm gets its name because smaller elements "bubble" to the top during each pass.

Time Complexity:

• Best Case (Already Sorted): O(n)

• Average Case: O(n²)

• Worst Case (Reversely Sorted): O(n²)

**Algorithm:**

1. Start with an array of size n.

2. Loop i from 0 to n-1.

3. Inside this loop, loop j from 0 to n-i-1.

4. Compare the adjacent elements; if the left one is bigger than the right one, swap them.

5. Repeat until no swaps are needed.

6. The array is sorted.

**Source Code (C++):**

#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];

}

// Bubble Sort with swap tracking

for (int i = 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;

}

}

// Print the array after each pass

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 1 2

Pass 1:

Swapping 5 and 3

Swapping 5 and 1

Swapping 5 and 2

Array after pass 1: 3 5 1 2 8

Pass 2:

Swapping 5 and 1

Swapping 5 and 2

Array after pass 2: 3 1 2 5 8

Pass 3:

Swapping 3 and 1

Swapping 3 and 2

Array after pass 3: 1 2 3 5 8

Pass 4:

No swaps, already sorted.

Final sorted array: 1 2 3 5 8

**Conclusion:**

1. The Bubble Sort algorithm was successfully implemented in C++.
2. The algorithm sorts an array by repeatedly swapping adjacent elements.

**Experiment No:** 2

**Name of the Experiment:**

A program to find an Element using a Linear Search algorithm.

**Objectives:**

1.Familiarization with the Linear Search algorithm for search.

2.To implement the Linear Search algorithm in C++.

3.Familiarization with the time complexity of Linear Search.

**Theory:**

Linear Search is a simple searching algorithm that compares each element in the list one by one until the target element is found or the end of the list is encountered. It is the most straightforward searching technique, ideally suited for small lists or unordered data.

**Time Complexity:**

•Best Case (Element at the beginning): O(1)

•Average Case: O(n)

•Worst Case (Element not present or at the end): O(n)

**Algorithm:**

1.Accept the size of the array from the user.

2. Accept array elements as input from the user.

3. Accept the element to be searched as input.

4. Search the array from beginning.

5. Compare each element with the target element.

6. Return the index if the element is found.

7. If the end of the array is reached without the element, return -1.

**Source Code (C++):**

#include <iostream>

using namespace std;

int main() {

int n, key;

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]; }

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

cin >> key;

// Linear Search

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

if (arr[i] == key) {

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

return 0; // Exit if found }

}

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

return 0;}

**Output:**

Enter the size of the array: 5

Enter 5 elements: 10 20 30 40 50

Enter the element to search: 30

Element found at index 2

**Conclusion:**

1. The Linear Search algorithm was successfully implemented in C++.
2. The algorithm searches for an element by checking each entry sequentially.
3. Linear Search is simple but inefficient for large datasets compared to binary search or hashing.

**Experiment No: 3**

**Name of the Experiment:**

A program to sort a linear array using the Merge Sort algorithm.

**Objectives:**

1. To understand the concept of sorting using the Merge Sort algorithm.

2. To implement the Merge Sort algorithm in C++.

3. To analyze the time complexity of Merge Sort.

**Theory:**

Merge Sort is a divide-and-conquer algorithm that divides an array into two halves, recursively sorts both halves, and then merges the sorted halves back. This provides a stable and efficient sorting process.

Time Complexity:

•Best Case: O(n log n)

•Average Case: O(n log n)

•Worst Case: O(n log n)

**Algorithm:**

1.If the array contains one or zero elements, then it is already sorted.

2.Divide the array into two halves.

3.Recursively sort both halves.

4. Merge the sorted halves again.

5. Continue merging until the whole array is sorted.

**Source Code (C++):**

#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 i = 0; i < n2; i++)

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

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);

}

}

int main() {

int n;

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

cin >> n;

int arr[n];

cout << "Enter the elements: ";

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

cin >> arr[i];

mergeSort(arr, 0, n - 1);

cout << "Sorted array: ";

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

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

cout << endl;

return 0;

}

**Output:**

Enter the number of elements: 5  
Enter the elements: 38 27 43 3 9  
Sorted array: 3 9 27 38 43

**Conclusion:**

1. The Merge Sort algorithm was successfully implemented in C++.
2. The algorithm recursively divides the array and merges sorted halves.
3. Merge Sort is more efficient than Bubble Sort and works well for large datasets.

**Experiment No: 4**

**Name of the Experiment:**

A program to find an element using the Binary Search algorithm.

**Objectives:**

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

**Theory:** Binary Search is a search algorithm that determines the location of a target element within a sorted array. It proceeds by continuously halving the search interval. If the target value is less than the middle element, the search proceeds in the left subarray; otherwise, it proceeds in the right subarray. This continues until the element is located or the search interval is empty.

**Time Complexity:**

Best Case: O(1) (Element is at the middle)

Average Case: O(log n)

Worst Case: O(log n)

**Algorithm:**

1. Take the sorted array and the target element as parameters.
2. Create two pointers, low at the start and high at the end of the array.
3. Compute the middle index mid as (low + high) / 2.
4. If arr[mid] equals the target element, return mid.
5. If arr[mid] is greater than the target, update high = mid - 1.
6. If arr[mid] is smaller than the target, set low = mid + 1.
7. Repeat steps 3-6 until low > high.
8. If the element is not found, return -1.

**Source Code (C++):**

#include <iostream>

using namespace std;

int main() {

int n, key;

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

cin >> n;

int arr[n];

cout << "Enter the sorted elements: ";

for (int i = 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: 5  
Enter the sorted elements: 10 20 30 40 50  
Enter the element to search: 30  
Element found at index: 2

**Conclusion:**

1. The Binary Search algorithm was successfully implemented in C++.
2. The algorithm efficiently searches for an element by dividing the array into halves.
3. Binary Search is much faster than Linear Search for sorted datasets, with a time complexity of O(log n).

**Experiment No: 5**

**Name of the Experiment:**

Program to search for a given pattern from text using the Pattern Matching algorithm**.**

**Objectives:**

1.To develop an appreciation of the definition of pattern matching for strings.

2.To create a simple Pattern Matching algorithm in C++.

3.To analyze the pattern matching algorithm in terms of its efficiency.

**Theory:**

Pattern Matching is a basic string processing operation, i.e., searching for a given fixed substring (pattern) in the input text. Naive String Matching Algorithm is the most obvious pattern matching algorithm, and it checks at every position in the text whether the pattern or not. We have other more optimized algorithms like KMP (Knuth-Morris-Pratt) and Boyer-Moore algorithms which try to optimize the search by excluding useless comparisons.

**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 (C++):**

#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: hello world, welcome to DSA  
Enter the pattern to search: world  
Pattern found at index 6

**Conclusion:**

1. The Pattern Matching program was successfully implemented in C++.

2. The Naive method examines every position of the text in order.

3. Further sophisticated algorithms like KMP and Boyer-Moore can refine the search further.

**Experiment No:** 6  
**Name of the Experiment:**

A program to implement a Queue data structure along with its typical operations.

**Objectives:**

1. To understand the concept of a queue data structure.
2. To implement a queue using an array in C++.
3. To perform basic operations such as enqueue, dequeue, and display.

**Theory:**

A queue is a linear data structure that follows the **FIFO (First In, First Out)** principle. Elements are inserted from the **rear** and removed from the **front**. A queue can be implemented using arrays or linked lists. Common operations include:

* **Enqueue**: Insert an element at the rear of the queue.
* **Dequeue**: Remove an element from the front of the queue.
* **Peek**: Retrieve the front element without removing it.
* **isEmpty**: Check if the queue is empty.
* **isFull**: Check if the queue is full (for array implementation).

**Time Complexity:**

* Enqueue: O(1)
* Dequeue: O(1)
* Peek: O(1)
* isEmpty / isFull: O(1)

**Algorithm:**

1. Initialize an empty queue with front = -1 and rear = -1.
2. **Enqueue operation:**
   * If the queue is full, display an overflow message.
   * Else, insert the element at rear and increment rear.
3. **Dequeue operation:**
   * If the queue is empty, display an underflow message.
   * Else, remove the element from front and increment front.
4. **Display operation:**
   * Print all elements from front to rear.

**Source Code (C++):**

#include <iostream>

using namespace std;

#define SIZE 5

int arr[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 (int i = 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.

1. Enqueue 20

Inserted 20 into the queue.

1. Enqueue 30

Inserted 30 into the queue.

3. Display

Queue elements: 10 20 30

2. Dequeue

Removed 10 from the queue.

3. Display

Queue elements: 20 30

4. Exit

**Conclusion:**

1. The queue data structure was successfully implemented using an array in C++.
2. Basic operations such as enqueue, dequeue, and display were performed without functions.
3. The queue follows the FIFO principle, making it suitable for scheduling tasks and handling requests.

**Experiment No: 7**

**Name of the Experiment:**

A program to solve the N-Queens problem using backtracking.

**Objectives:**

1. To know the concept of backtracking.

2. To solve the N-Queens problem using backtracking in C++.

3. To find all the possible ways to place N queens on an N×N chessboard such that no two queens attack each other.

**Theory:**

N-Queens problem is a classic combinatorics problem that consists of placing N queens on an N×N chessboard in such a manner that no queen is capable of attacking any other. That is:

•No two queens are permitted in the same row.

•No two queens are permitted in the same column.

•No two queens are permitted in the same diagonal.

**Backtracking Approach:**

Backtracking is the technique of trying all feasible positions by placing a queen at a time in every column and backtracking if any placement yields an invalid solution. The steps are:

1.\tStart with the leftmost column.

2.\tTry placing a queen in each row of the present column.

3.\tIf there exists a safe spot, place the queen and try the next column.

4.\tIf the placement of all queens is made, print the solution.

5. If that leads to no solution, then backtrack to the previous column and place the queen in some other position.

**Algorithm:**

1. Write a function to determine if placing a queen at board[row][col] is safe.

2. If all the queens are placed in all the columns, then display the board.

3. Try placing a queen in each row of the current column and recursively search for a solution for the next column.

4. If the placement of a queen results in a solution, return true. Otherwise, delete the queen (backtrack) and attempt another row.

5. If no placement is successful, return false.

**Source Code (C++):**

#include <iostream>

using namespace std;

#define N 8

int board[N][N] = {0};

bool isSafe(int row, int col) {

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

if (board[row][i])

return false;

for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)

if (board[i][j])

return false;

for (int i = row, j = col; i < N && j >= 0; i++, j--)

if (board[i][j])

return false;

return true;

}

bool solveNQueens(int col) {

if (col >= N)

return true;

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

if (isSafe(i, col)) {

board[i][col] = 1;

if (solveNQueens(col + 1))

return true;

board[i][col] = 0;

}

}

return false;

}

void printBoard() {

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

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

cout << board[i][j] << " ";

cout << endl;

}

}

int main() {

if (solveNQueens(0))

printBoard();

else

cout << "No solution exists" << endl;

return 0;

}

**Output:**

Example output for N=8:

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

**Conclusion:**

1. The N-Queens problem was successfully solved using the backtracking approach.
2. The algorithm explores all possibilities and backtracks when necessary to find the correct placements.
3. The implementation can be extended to different values of N by modifying the N constant in the program.

**Experiment No:** 8  
**Name of the Experiment:**

A program to solve the Sum of Subset problem.

**Objectives:**

1. To understand the principle of backtracking in solving combinatorial problems.

2. Solving the Sum of Subset problem without using functions.

3. To create each potential subset whose sum is equal to a given target value, according to user input.

**Theory:** The Sum of Subset problem is a traditional classic combinatorial optimization and backtracking problem. You are given a set of positive integers and a target sum d. You are supposed to find all of the subsets of the set with sums equal to d.

Retrospective Approach:

Backtracking is utilized to form all subsets while making sure unnecessary checks are prevented if the sum is greater than d. The operations are:

1. Read the input set and target sum from input.

2. Write a loop-based solution to generate subsets iteratively.

3. If the sum of the subset is d, print the subset.

4. Keep looking for other potential subsets.

**Algorithm:**

1. Accept the user input for the set elements and the target sum.

2. Generate all possible subsets using a loop-based approach.

3. If the sum of the selected subset is d, then print the subset.

4. If the sum exceeds d, discard that subset.

5. Repeat the procedure until all alternatives are tried.

**Source Code (C++):**

#include <iostream>

using namespace std;

int main() {

int n, d;

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

cin >> n;

int set[n];

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

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

cin >> set[i];

}

cout << "Enter the target sum: ";

cin >> d;

int totalSubsets = 1 << n;

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

for (int mask = 0; mask < totalSubsets; mask++) {

int sum = 0;

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

if (mask & (1 << j)) {

sum += set[j];

}

}

if (sum == d) {

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

if (mask & (1 << j)) {

cout << set[j] << " ";

}

}

cout << endl;

}

}

return 0;

}

**Output:**

Enter number of elements in the set: 4

Enter the elements of the set: 3 1 2 5

Enter the target sum: 6

Subsets with sum 6 are:

3 1 2

1 5

**Conclusion:**

1.The Sum of Subset problem was successfully implemented without using functions.

2. The algorithm efficiently detects all subsets summing up to the target sum.

3. The approach is also valid for more extensive data series and different target totals via input modification.

**Experiment No:** 9  
**Name of the Experiment:**

A program to solve the 0/1 Knapsack problem using dynamic programming.

**Objectives:**

1. To understand the concept of the **0/1 Knapsack problem** in combinatorial optimization.
2. To implement the **dynamic programming** approach for solving the problem efficiently.
3. To determine the maximum profit achievable within the given knapsack capacity.

**Theory:**

The **0/1 Knapsack problem** is a well-known problem in dynamic programming where we have a set of items, each with a **weight** and **profit**. The goal is to determine the maximum total profit that can be obtained by selecting a subset of items, ensuring that their total weight does not exceed the given capacity. Each item can either be included or excluded (hence the name **0/1 Knapsack**).

**Dynamic Programming Approach:**

1. **Define the DP State:** Let dp[i][w] represent the maximum profit obtained by selecting from the first i items with a knapsack capacity of w.
2. **Recurrence Relation:**
   * If we exclude item i: dp[i][w] = dp[i-1][w]
   * If we include item i: dp[i][w] = profit[i] + dp[i-1][w - weight[i]] (if weight[i] ≤ w)
   * Take the maximum of both cases:

dp[i][w] = max(dp[i-1][w], profit[i] + dp[i-1][w - weight[i]])

1. **Base Case:** If i == 0 or w == 0, then dp[i][w] = 0.

**Algorithm:**

1. Take input from the user for the number of items, profits, weights, and knapsack capacity.
2. Create a DP table dp[n+1][C+1] initialized to zero.
3. Fill the table using the recurrence relation.
4. The maximum profit is stored in dp[n][C].
5. Print the maximum profit.

**Source Code (C++):**

#include <iostream>

using namespace std;

int main() {

int n, C;

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

cin >> n;

int profit[n], weight[n];

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

for (int i = 0; i < n; i++) cin >> profit[i];

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

for (int i = 0; i < n; i++) cin >> weight[i];

cout << "Enter the knapsack capacity: ";

cin >> C;

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

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

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

if (i == 0 || w == 0)

dp[i][w] = 0;

else if (weight[i-1] <= w)

dp[i][w] = max(dp[i-1][w], profit[i-1] + dp[i-1][w - weight[i-1]]);

else

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

}

}

cout << "Maximum profit: " << dp[n][C] << endl;

return 0; }

**Output:**

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 knapsack capacity: 20

Maximum profit: 40

**Conclusion:**

1. The **0/1 Knapsack** problem was successfully solved using dynamic programming.
2. The approach efficiently calculates the maximum profit within the given constraints.
3. The algorithm ensures **optimal substructure** and **overlapping subproblems**, making it an ideal case for dynamic programming.

**Experiment No:** 10  
**Name of the Experiment:**

A program to solve the Tower of Hanoi problem for the N disk.

**Objectives:**

1. To understand the concept of the **Tower of Hanoi** problem.
2. To implement a recursive approach to solve the problem.
3. To analyze the time complexity of the Tower of Hanoi algorithm.

**Theory:**

The **Tower of Hanoi** is a mathematical puzzle consisting of three rods and N disks of different sizes. The puzzle starts with all the disks stacked in **increasing order of size** on one rod, and the goal is to move all the disks to another rod while obeying the following 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 any rod can be moved.

The **recursive approach** solves the problem as follows:

* Move the top N-1 disks from **Source** to **Auxiliary** rod.
* Move the largest disk from **Source** to **Destination** rod.
* Move the N-1 disks from **Auxiliary** to **Destination** rod.

**Algorithm:**

1. Take input from the user for the number of disks.
2. Use recursion to move N-1 disks to the auxiliary rod.
3. Move the largest disk to the destination rod.
4. Move the N-1 disks from the auxiliary rod to the destination rod.
5. Display the sequence of moves.

**Source Code (C++):**

#include <iostream>

using namespace std;

int main() {

int N;

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

cin >> N;

int moves = (1 << N) - 1; // Total moves required (2^N - 1)

int from[3] = {1, 2, 3}; // Rods

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

int disk = \_\_builtin\_ctz(i) + 1; // Find the disk to be moved

int src = (i & i - 1) % 3;

int dest = ((i | i - 1) + 1) % 3;

cout << "Move disk " << disk << " from rod " << from[src] << " to rod " << from[dest] << endl;

}

return 0;

}

**Output:**

Enter the number of disks: 3

Move disk 1 from rod 1 to rod 3

Move disk 2 from rod 1 to rod 2

Move disk 1 from rod 3 to rod 2

Move disk 3 from rod 1 to rod 3

Move disk 1 from rod 2 to rod 1

Move disk 2 from rod 2 to rod 3

Move disk 1 from rod 1 to rod 3

**Conclusion:**

1. The **Tower of Hanoi** problem was successfully solved using recursion.
2. The problem follows an **exponential time complexity** of O(2^N).
3. The recursive approach provides an elegant and structured solution for solving the Tower of Hanoi puzzle.