**PABNA UNIVERSITY OF SCIENCE AND TECHNOLOGY**

Date of Issue: Date of Submission: 01-03-2025

Submitted To:

**Md. Anwar Hossain**

Professor

Department of ICE

Pabna University of Science and Technology, Pabna.

Submitted By:

**Mahmudul Islam**

Roll: 220639

Session: 2021-2022

2nd Year 2nd Semester

Department of ICE, PUST

**Faculty of Engineering & Technology**

**Department of Information and Communication Engineering**

**LAB REPORT**

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

**Course Code: ICE - 2202**

|  |  |
| --- | --- |
| **Sl.** | **Description Problem** |
| **01** | **Write a program to sort a linear array using the bubble sort algorithm.** |
| **02** | **Write a program to find an element using a linear search algorithm.** |
| **03** | **Write a program to sort a linear array using the merge sort algorithm.** |
| **04** | |  | | --- | |  |   **Write a program to find an element using the binary search algorithm.**   |  | | --- | |  | |
| **05** | **Write a program to find a given pattern from text using the pattern matching algorithm.** |
| **06** | **Write a program to implement a queue data structure along with its typical operations.** |
| **07** | **Write a program to solve n queen's problem using backtracking.** |
| **08** | **Consider a set S = {5, 10, 12, 13, 15, 18} and d = 30. Write a program to solve the sum of subset problem.** |
| **09** | **Write a program to solve the following 0/1 Knapsack using dynamic programming approach: Profits P = (15, 25, 13, 23), Weights W = (2, 6, 12, 9), Knapsack Capacity C = 20, and the number of items n = 4.** |

**Index**

|  |  |
| --- | --- |
| **10** | **Write a program to solve the Tower of Hanoi problem for N disks.** |

**Problem Number: 1**

**Title : Bubble Sort Implementation for Linear Array**

**Illustration of the Problem:**  
Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process repeats until the entire list is sorted. The algorithm gets its name because smaller elements "bubble" to the top of the list with each pass.

**Algo8rithm:**

1. Start by iterating through the array from the beginning.
2. Compare each pair of adjacent elements.
3. If the first element is greater than the second, swap them.
4. Continue this process for the entire array.
5. Repeat steps 1-4 until no swaps are required, indicating that the array is sorted.

**Source Code:**

**#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;**

**}**

**Sample Input and Output:** **Input:**

Enter the number of elements in the array: 5

Enter the elements of the array: 64 34 25 12 22

**Output:**

Sorted array in ascending order: 12 22 25 34 64

**Problem Number: 2**

**Title: Linear Search Algorithm for finding an Element**

**Illustration of the Problem:**

The objective of this problem is to find an element within a given array using the **Linear Search** algorithm. Linear search is a simple searching technique where each element of the array is compared with the target value sequentially until a match is found or the array is completely traversed.

**Logic of the Solution:**

1. **Input**: The user provides an integer array and a key element to search for.
2. **Process**:
   * Traverse the array sequentially from the first to the last element.
   * At each step, check if the current element is equal to the key.
   * If a match is found, the index of the element is displayed, and the search is terminated.
   * If no match is found after checking all elements, display a message indicating that the element is not in the array.
3. **Output**: The position of the element in the array if found; otherwise, a message indicating it was not found.

**Step-by-step Procedure (Algorithm):**

1. Start.
2. Take the input for the number of elements n in the array.
3. Initialize the array and take the input for the array elements.
4. Take input for the key element to search.
5. Initialize a flag variable found to 0 (indicating element not found initially).
6. Loop through each element of the array:
   * If the element at the current index matches the key:
     + Print the index of the element.
     + Set found to 1 and break the loop.
7. If found remains 0 after the loop:
   * Print a message indicating that the key element was not found.
8. End.

**Source Code:**

**#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;**

**}**

**Sample Input and Output:**

Enter the number of elements in the array: 5

Enter the elements of the array: 12 34 7 56 23

Enter the element to search: 7

Element 7 found at index 2.

**Problem Number: 3**

**Title: Merge Sort Algorithm for Sorting a Linear Array**

**Illustration of the Problem:**

The task is to sort a given array using the **Merge Sort** algorithm. Merge Sort is an efficient, divide-and-conquer algorithm that splits the array into two halves, sorts them recursively, and then merges them back into a sorted array. This method ensures a time complexity of O(n log n) for sorting the array.

**Logic of the Solution:**

* **Divide**: The array is split into two halves recursively until each half contains only one element.
* **Conquer**: Each of these halves is sorted individually. Since a single element is trivially sorted, this step involves recursively sorting smaller arrays.
* **Combine**: After sorting the halves, they are merged back into a single sorted array. The merging step ensures that elements are in ascending order.

The recursive nature of the algorithm allows it to handle large arrays efficiently.

**Step-by-step Procedure (Algorithm):**

1. Start.
2. **Base Case**: If the size of the array is 1 (i.e., left >= right), the array is already sorted, so return.
3. **Divide**: Find the middle of the array and split it into two halves:
   * Left half: from index left to mid
   * Right half: from index mid + 1 to right
4. **Recursion**: Recursively sort the left and right halves of the array.
5. **Merge**: After sorting the two halves, merge them into a single sorted array:
   * Compare the smallest unmerged elements of both halves and place the smaller one in the sorted array.
   * Repeat this process until all elements from both halves are merged.
6. Output the sorted array.
7. End.

**Source Code:**

**#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;**

**}**

**Sample Input and Output:**

Enter the number of elements in the array: 6

Enter the elements of the array: 34 7 23 32 5 62

Sorted array: 5 7 23 32 34 62

**Problem Number: 4**

**Title: Binary Search Algorithm for Finding an Element in a Sorted Array**

**Illustration of the Problem:**

The task is to find an element in a sorted array using the **Binary Search** algorithm. Binary Search is an efficient algorithm that repeatedly divides the search interval in half. If the value of the search key is less than the item in the middle of the interval, the search continues in the lower half, otherwise, it continues in the upper half. This process repeats until the target element is found or the search interval is empty.

**Logic of the Solution:**

* The array must be sorted for binary search to work.
* Start with two pointers, left and right, at the ends of the array.
* Find the middle element of the array.
* If the middle element matches the target, return its index.
* If the target is smaller than the middle element, search the left half.
* If the target is larger than the middle element, search the right half.
* Repeat the process until the element is found or the search range is exhausted.

**Step-by-step Procedure (Algorithm):**

1. Start.
2. Define the range of the search with two pointers, left and right, pointing to the start and end of the array.
3. While the left pointer is less than or equal to the right pointer:
   * Calculate the middle index mid as (left + right) / 2.
   * If the middle element equals the target, return the index.
   * If the target is smaller, adjust right to mid - 1 to search the left half.
   * If the target is larger, adjust left to mid + 1 to search the right half.
4. If the element is not found, return -1 indicating that the target is not in the array.
5. End.

**Source Code:**

**#include <iostream>**

**using namespace std;**

**int binarySearch(int arr[], int n, int key) {**

**int left = 0, right = n - 1;**

**while (left <= right) {**

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

**if (arr[mid] == key) {**

**return mid; // Return the index of the key**

**}**

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

**right = mid - 1;**

**}**

**else {**

**left = mid + 1;**

**}**

**}**

**return -1;**

**}**

**int main() {**

**int n, key;**

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

**cin >> n;**

**int arr[n];**

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

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

**cin >> arr[i];**

**}**

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

**cin >> key;**

**int result = binarySearch(arr, n, key);**

**if (result != -1) {**

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

**} else {**

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

**}**

**return 0;**

**}**

**Sample Input and Output:**

Enter the number of elements in the array: 6

Enter the elements of the sorted array: 5 7 23 32 34 62

Enter the element to search: 23

Element 23 found at index 2

**Problem Number: 5**

**Title: Pattern Matching Algorithm for Finding a Pattern in a Text**

**Illustration of the Problem:**

The task is to find a given pattern in a text using the **Pattern Matching** algorithm. The algorithm checks whether the pattern is present in the given text by comparing characters one by one. If the pattern matches, the algorithm returns the index where the pattern starts in the text.

**Logic of the Solution:**

1. Start from the beginning of the text and compare the first character of the pattern with the corresponding character of the text.
2. If the characters match, continue comparing the rest of the characters in the pattern with the characters in the text.
3. If all characters match, the pattern is found at the current position in the text.
4. If any character does not match, move to the next position in the text and repeat the process.
5. If no match is found after checking the entire text, return that the pattern is not found.

**Step-by-step Procedure (Algorithm):**

1. Start.
2. Iterate through the text from the beginning to the point where the pattern can still fit.
3. For each position in the text, compare the pattern with the substring starting at that position.
4. If the entire pattern matches, return the index where the pattern starts.
5. If the pattern is not found, return -1 indicating no match.
6. End.

**Source Code:**

#include <iostream>

using namespace std;

int patternMatch(string text, string pattern) {

int n = text.length();

int m = pattern.length();

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

int j = 0;

while (j < m && text[i + j] == pattern[j]) {

j++;

}

if (j == m) {

return i;

}

}

return -1;

}

int main() {

string text, pattern;

cout << "Enter the text: ";

getline(cin, text);

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

getline(cin, pattern);

int result = patternMatch(text, pattern);

if (result != -1) {

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

} else {

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

}

return 0;

}

**Sample Input and Output:**  
Enter the text: hello world

Enter the pattern to search: world

Pattern found at index 6

**Problem Number: 6**

**Title: Queue Data Structure Implementation with operations**

**Illustration of the Problem:**

The task is to implement a queue data structure with its typical operations: enqueue (inserting an element), dequeue (removing an element), and display the elements of the queue. The queue is a **First-In-First-Out (FIFO)** data structure, meaning the element inserted first is the one removed first.

**Logic of the Solution:**

1. **Enqueue**: Insert an element at the end of the queue. If the queue is full, an error message is displayed.
2. **Dequeue**: Remove an element from the front of the queue. If the queue is empty, an error message is displayed.
3. **Display**: Show all the elements of the queue from front to rear.

We will implement these operations in separate functions for better organization and reuse.

**Step-by-step Procedure (Algorithm):**

1. **Enqueue Operation**:
   * Check if the queue is full. If full, print an error message.
   * If not, add the element to the rear of the queue.
2. **Dequeue Operation**:
   * Check if the queue is empty. If empty, print an error message.
   * If not, remove the element from the front and shift the front pointer.
3. **Display Operation**:
   * Traverse the queue from front to rear and print all elements.

**Source Code:**

**#include <iostream>**

**using namespace std;**

**#define MAX 5**

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

**void enqueue(int value) {**

**if (rear == MAX - 1) {**

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

**} else {**

**if (front == -1) {**

**front = 0;**

**}**

**rear++;**

**queue[rear] = value;**

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

**}**

**}**

**void dequeue() {**

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

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

**} else {**

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

**front++;**

**if (front > rear) {**

**front = rear = -1;**

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

**}**

**}**

**}**

**void display() {**

**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;**

**}**

**}**

**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:**

**cout << "Enter the value to enqueue: ";**

**cin >> value;**

**enqueue(value);**

**break;**

**case 2:**

**dequeue();**

**break;**

**case 3:**

**display();**

**break;**

**case 4:**

**cout << "Exiting...\n";**

**return 0;**

**default:**

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

**}**

**}**

**return 0;**

**}**   
**Sample Input and Output:**

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 1

Enter the value to enqueue: 10

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 1

Enter the value to enqueue: 20

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 3

Queue elements are: 10 20

**Problem Number: 7**

**Title: N-Queens Problem using Backtracking**

**Illustration of the Problem:**

The **N-Queens Problem** is a classic combinatorial problem where the goal is to place **N queens** on an **N×N chessboard** such that:

1. No two queens attack each other.
2. Queens cannot be placed in the same row, column, or diagonal.

We use **backtracking** to explore possible placements recursively and revert when a conflict arises.

**Step-by-Step Procedure (Algorithm):**

1. Start placing queens row by row from row 0.
2. For each row, try placing a queen in each column.
3. Before placing, check if the queen is safe from attacks:
   * No other queen in the same column.
   * No queen on the left diagonal.
   * No queen on the right diagonal.
4. If it is safe, place the queen and move to the next row.
5. If all queens are placed successfully, print the solution.
6. If no column is safe in the current row, **backtrack** and try a different position in the previous row.
7. Repeat until all solutions are found.

**Source Code**

#include <iostream>

using namespace std;

#define N 4

void printSolution(int board[N][N]) {

static int count = 0;

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

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

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

cout << (board[i][j] ? "Q " : ". ");

}

cout << endl;

}

}

bool isSafe(int board[N][N], int row, int col) {

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

if (board[i][col]) 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 >= 0 && j < N; i--, j++)

if (board[i][j]) return false;

return true;

}

bool solveNQueens(int board[N][N], int row) {

if (row == N) { // If all queens are placed, print solution

printSolution(board);

return true;

}

bool foundSolution = false;

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

if (isSafe(board, row, col)) {

board[row][col] = 1; // Place queen

foundSolution |= solveNQueens(board, row + 1); // Recur for next row

board[row][col] = 0; // Backtrack

}

}

return foundSolution;

}

int main() {

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

if (!solveNQueens(board, 0)) {

cout << "No solution exists.\n";

}

return 0;

}   
Sample Output for N = 4   
Solution 1:

Q . . .

. . Q .

. . . Q

. Q . .

Solution 2:

. Q . .

. . . Q

Q . . .

. . Q .

**Problem Number : 8**

**Title : Sum of Subsets: Finding Subsets with Target Sum**

**Illustration of the Problem:**  
The Sum of Subset problem is a classic combinatorial problem where we need to determine all subsets of a given set whose sum equals a specified target sum. This problem can be solved using a bit-masking approach to generate all possible subsets and check their sums.

**Algorithm:**

1. Read the number of elements in the set.
2. Input the elements of the set.
3. Read the target sum value.
4. Generate all possible subsets using bit manipulation.
5. Check if the sum of a subset equals the target sum.
6. If a subset satisfies the condition, print it.
7. Count and display the number of valid subsets.

**Source Code:**

**#include <iostream>**

**#include <cmath>**

**using namespace std;**

**void findSubsets(int S[], int N, int target\_sum) {**

**int total\_subsets = 1 << N;**

**int count = 0;**

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

**int subset\_sum = 0;**

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

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

**subset\_sum += S[j];**

**}**

**}**

**if (subset\_sum == target\_sum) {**

**cout << "{ ";**

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

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

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

**}**

**}**

**cout << "}\n";**

**count++;**

**}**

**}**

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

**}**

**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;**

**findSubsets(S, N, target\_sum);**

**return 0;**

**}**   
**Sample Input and Output:** **Input:**

Enter the number of elements: 6

Enter the elements: 5 10 12 13 15 18

Enter the target sum: 30

{ 5 10 15 }

{ 5 12 13 }

{ 12 18 }

Total subsets found: 3   
  
  
**Problem Number :9**

**Title** **: 0/1 Knapsack Problem using Dynamic Programming**

**Problem Illustration**

The 0/1 Knapsack problem is a combinatorial optimization problem where we have a set of items, each with a given weight and profit. The goal is to determine the most profitable subset of items to include in a knapsack of a given capacity such that the total weight does not exceed the capacity.

Given:

* Profits, P = {11, 21, 31, 33}
* Weights, W = {2, 11, 22, 15}
* Knapsack Capacity, C = 40
* Number of Items, n = 4

We use dynamic programming to find the optimal subset of items to maximize the profit while staying within the weight limit.

**Step-by-Step Procedure**

1. Define a 2D DP table dp[i][j] where i represents the number of items considered and j represents the weight limit of the knapsack.
2. Initialize the first row and first column of the DP table with 0 (when no items or weight is 0).
3. Use the recurrence relation:
   * If an item's weight is less than or equal to j,
   * Otherwise, exclude the item:
4. Construct the optimal subset of items by backtracking through the DP table.
5. Output the selected items and the maximum profit.

**Source Code :**

#include <iostream>

using namespace std;

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

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

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

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

if (i == 0 || j == 0)

dp[i][j] = 0;

else if (W[i-1] <= j)

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

else

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

}

}

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

int w = C;

int x[n] = {0};

for (int i = n; i > 0 && w > 0; i--) {

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

x[i-1] = 1;

w -= W[i-1];

}

}

cout << "Solution Vector: ";

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

cout << x[i] << " ";

}

cout << endl;

}

int main() {

int P[] = {11, 21, 31, 33};

int W[] = {2, 11, 22, 15};

int C = 40;

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

knapsack(P, W, n, C);

return 0;

}

**Sample Input & Output**

Profits: {11, 21, 31, 33}

Weights: {2, 11, 22, 15}

Knapsack Capacity: 40

Number of Items: 4

Maximum Profit: 65

Solution Vector: 1 1 0 1

**Problem Number : 10**

**Title:** Tower of Hanoi Problem using Iterative Approach

**Problem Illustration:** The Tower of Hanoi is a mathematical puzzle that involves moving N disks from a source peg to a destination peg using an auxiliary peg while following these rules:

1. Only one disk can be moved at a time.
2. A disk can only be placed on an empty peg or on top of a larger disk.
3. All disks start on the source peg and must be moved to the destination peg.

Given: Number of Disks, N Source Peg, Destination Peg, Auxiliary Peg

**Step-by-Step Procedure:**

1. Calculate the total number of moves required using the formula 2^N - 1.
2. Use a stack-based iterative approach to store and process each move.
3. Push the initial problem (N disks) onto the stack.
4. While the stack is not empty:
   * Pop the top element and retrieve disk count, source peg, destination peg, and auxiliary peg.
   * If only one disk needs to be moved, print the move.
   * Otherwise, push the subproblems onto the stack in the correct order to maintain move sequence.
5. Continue until all moves are executed and the destination peg contains all disks.

**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;**

**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;**

**}**   
**Sample Input and Output:**

Enter the number of disks: 3

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

Total number of moves: 7

A -> C

A -> B

C -> B

A -> C

B -> A

B -> C

A -> C