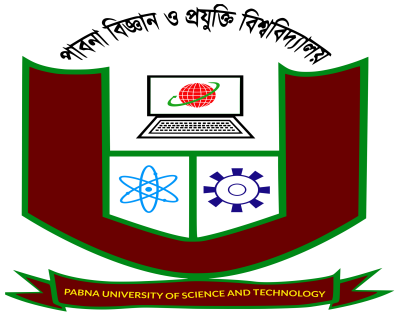
**PABNA UNIVERSITY OF SCIENCE AND TECHNOLOGY**



LAB REPORT

Course Title:Data Structure & Algorithm sessional

Course Code:ICE-2202

Submitted By: Submitted To:

Shakil Ahmed Md. Anwar Hossain

Roll : 220636 Chairman of

Session:2021-22 Department of Information

Department of Information & Communication

& Coummunication Engineering

Sumission date:26-02-2025

Index

|  |  |
| --- | --- |
| SL. | Problem Description |
| 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 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), weight W=(2,6,12,9), knapsack C=20, and the number of items n=4.** |
| 10 | **Write a program to solve the Tower of Hanoi problem for the N disk.** |

**Problem 1 :Write a program to sort a linear array using the bubble sort algorithm.**

**Logic :Logic of Bubble Sort:**

1. **Initialization:** Start with an unsorted array.
2. **Outer Loop:** Repeat the process for the entire array length — after each pass, the largest unsorted element will be in its correct position at the end.
3. **Inner Loop:** Compare adjacent elements. If the current element is greater than the next, swap them. This pushes the larger element toward the end.
4. **Swapping:** Keep track of whether any swaps occurred during the pass. If no swaps happen in a full pass, the array is already sorted — exit early.
5. **Optimization:** With each iteration, the end of the array becomes sorted, so the inner loop shortens its range.
6. **Completion:** Once no swaps are needed, the array is fully sorted.

**Algorithm:Bubble Sort Algorithm:**

1. Start from the first element of the array.
2. Compare the current element with the next element.
3. If the current element is greater than the next one, swap them.
4. Move to the next pair of elements and repeat steps 2 and 3 for the entire array.
5. After each pass, the largest element "bubbles up" to the end of the array.
6. Repeat the process for the remaining elements, ignoring the last sorted elements.
7. Stop when no more swaps are needed (the array is sorted)

#include<iostream>

using namespace std;

void bubblesort(int arr[], int n) {

    for(int i = 0; i < n - 1; i++) {

        //bool swapped = false;

        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 << "iteration " << i + 1 << " - swap: ";

    }

}

    }

    }

void printarray(int arr[], int n) {

    for(int i = 0; i < n; i++) {

        cout << arr[i] << " ";

    }

    cout << endl;

}

int main() {

    int n;

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

    cin >> n;

    int arr[n];

    cout << "Enter the array elements: ";

    for(int i = 0; i < n; i++) {

        cin >> arr[i];

    }

    cout << "Unordered array: ";

    printarray(arr, n);

    bubblesort(arr, n);

    cout << "Sorted array: ";

    printarray(arr, n);

    return 0;

}

**Input & Output :**

**Enter the number of elements in the array: 4 Enter the elements of the array: 5 7 2 4 Sorted array in ascending order: 2 4 5 7**

**Problem 2 :Write a program to find an element using a linear search algorithm.**

**Logic:**

* Linear search works by checking each element one by one.
* It’s simple and works on both sorted and unsorted arrays.
* **Time Complexity:** O(n) — because it may need to check every element.
* **Space Complexity:** O(1) — because it searches in-place

**Algorithm:**

1. Start from the first element of the array.
2. Compare the current element with the target element.
3. If they match, return the index of the element.
4. If not, move to the next element and repeat.
5. Continue until the end of the array.
6. If the element is not found, return an indicator like -1.

#include <iostream> using namespace std;

int main() {

int n, key, found = 0;

// Input the size of the array

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

int arr[n];

// Input the elements of the array

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

cin >> arr[i];

**}**

// Input the element to search

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

cin >> key;

// Linear Search Logic for (int i = 0; i < n; i++) {

if (arr[i] == key) {

cout << "Element " << key << " found at index " << i << "." << endl; found = 1;

break;

**}**

**}**

// If the element is not found if (!found) {

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

**}**

return 0;

**}**

**Input & Output :**

**Enter the number of elements in the array: 4 Enter the elements of the array: 5 7 8 9 Enter the element to search: 9**

**Element 9 found at index 3.**

**Problem 3 : write a program to sort a linear array using the merge sort algorithm.**

**Logic:**

* Merge sort is a **divide and conquer** algorithm.
* It keeps splitting the array until each sub-array has one element.
* Then it merges pairs of sorted arrays by comparing elements one by one.
* **Time Complexity:** O(n log n) (for all cases)
* **Space Complexity:** O(n) (because of additional arrays for merging)

**Algorithm:**

1. **Divide:** Split the array into two halves.
2. **Conquer:** Recursively sort each half.
3. **Combine:** Merge the sorted halves into a single sorted array.

#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 elemnets number of array : "; cin >> n;

int arr[n];

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

cin >> arr[i];

**}**

mergeSort(arr, 0, n - 1);

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

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

**}**

cout << endl;

return 0;

**}**

**Input &Output :**

**Enter the elemnets number of array : 4 Enter the elements of array : 8 4 3 6 The sorted array is : 3 4 6 8**

**Problem 4 : Write a program to find an element using the binary search algorithm.**

**Logic:**

* Binary search works on **sorted arrays**.
* It repeatedly divides the search space in half.
* By comparing the target element with the middle element, it eliminates half of the remaining elements in each step.
* This makes it far more efficient than linear search.

**Algorithm:**

1. **Initialize:** Set two pointers — low at the start and high at the end of the array.
2. **Loop:** While low <= high:  
   a. Find the middle element: mid = (low + high) // 2.  
   b. If arr[mid] equals the target, return mid.  
   c. If arr[mid] is less than the target, search the right half (low = mid + 1).  
   d. If arr[mid] is greater than the target, search the left half (high = mid - 1).
3. **Not Found:** If the loop ends, return -1 (element not in array).

**Time Complexity:** O(log n)  
**Space Complexity:** O(1) (for iterative) or O(log n) (for recursive

#include <iostream> using namespace std;

int main() {

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

//taking array size

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

int arr[n];

//taking sorted array

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

cin >> arr[i];

**}**

//searching number

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

// Binary Search Logic left = 0; right = n - 1;

while (left <= right) {

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

//printing the possition if (arr[mid] == target) {

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

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

} else {

right = mid - 1;

**}**

step++;

**}**

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

**}**

**Input & Output :**

**Enter the number of elements: 4 Enter 4 sorted elements: 3 5 6 9 Enter the number to search: 6 Target 6 found at position 3**

**Problem 5 :Write a program to find a given pattern from text using pattern matching algorithm.**

### ****Logic of Pattern Matching:****

* We check if a given pattern exists in a text.
* Start at each position in the text and compare the pattern with a substring of the same length.
* If all characters match, we have found the pattern.
* If not, move to the next position and repeat.
* This method works well for small patterns but is inefficient for large inputs.

### ****Algorithm:****

1. **Initialize:** Let n be the length of the text and m be the length of the pattern.
2. **Loop through the text:** Iterate from i = 0 to n - m.
3. **Compare characters:** For each i, check if text[i:i+m] matches the pattern.
4. **Pattern Found:** If a match is found, return the starting index.
5. **No Match:** If no match is found, return -1.

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

int main() {

char text[1000], pattern[1000];

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

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

int textLen = strlen(text);

int patternLen = strlen(pattern);

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

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

**}**

if (j == patternLen) {

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

**}**

**}**

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

**}**

**Input & Output :**

**Enter the text: hello world Enter the pattern: wor Pattern found at index 6**

**Problem 6 :Write a program to implement a queue data structure along with its typical operations.**

**Logic of Queue:**

* A **queue** is a **FIFO** (First In, First Out) data structure.
* Elements are added to the **rear** and removed from the **front**.

**Operations:**

1. **Enqueue:** Add an element to the rear of the queue.
2. **Dequeue:** Remove an element from the front of the queue.
3. **Peek/Front:** View the front element without removing it.
4. **IsEmpty:** Check if the queue is empty.
5. **Size:** Get the number of elements in the queue.

**Algorithm:**

1. **Enqueue:** Add the element to the end of the list.
2. **Dequeue:** Remove and return the first element of the list.
3. **Peek:** Return the first element without removing it.
4. **IsEmpty:** Check if the list’s length is 0.
5. **Size:** Return the list’s length.

#include <iostream> using namespace std;

#define MAX 5

int queue[MAX], front = -1, rear = -1; int main() {

int choice, value;

while (1) {

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

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

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

switch (choice) { case 1:

if (rear == MAX - 1) {

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

} else {

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

**}**

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

rear++;

queue[rear] = value;

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

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

**}**

break;

case 2:

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

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

} else {

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

front++;

cout << "Current front: " << front << ", rear: " << rear << endl; if (front > rear) {

front = rear = -1;

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

**}**

**}**

break;

case 3:

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

} else {

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

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

**}**

cout << endl;

**}**

break;

case 4:

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

default:

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

**}**

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

} else {

cout << "Current queue: ";

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

**}**

cout << endl;

**}**

**}**

return 0;

**}**

**Input & Output :**

**Queue Operations:**

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

**Enter your choice: 1**

**Enter the value to enqueue: 3 3 enqueued to the queue.**

**Current front: 0, rear: 0**

**Current queue: 3**

**Queue Operations:**

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

**Enter your choice: 1**

**Enter the value to enqueue: 4 4 enqueued to the queue.**

**Current front: 0, rear: 1**

**Current queue: 3 4**

**Queue Operations:**

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

**Enter your choice: 3**

**Queue elements are: 3 4 Current queue: 3 4**

**Queue Operations:**

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

**Enter your choice: 4 Exiting...**

**Problem 7 : Write a program to solve n queen’s problem using backtracking.**

### Logic:

The N-Queens problem involves placing N chess queens on an N×N chessboard such that no two queens can attack each other. This means that no two queens can share the same row, column, or diagonal.

To solve this problem using backtracking, we can use the following logic:

1. **Start placing queens**: Begin placing queens one by one, starting from the first row to the Nth row.
2. **Check if placement is safe**: For each cell in the current row, check if it's safe to place a queen there. The safety conditions are:
   1. The column must not already contain a queen.
   2. The diagonal (both directions) must not already contain a queen.
3. **Move to the next row**: If a queen can be placed in the current position, move on to the next row and attempt to place a queen there.
4. **Backtrack**: If placing a queen in any column of the current row leads to a conflict or doesn’t allow a solution, backtrack by removing the queen and trying the next column.
5. **End condition**: If all queens have been placed successfully, return the solution.

### Algorithm for Solving the N-Queens Problem Using Backtracking:

1. **Initialize** an empty N×N chessboard.
2. **Define a recursive function** solveNQueens(row) to attempt to place queens row by row.
   1. If row == N, the solution is found (all queens are placed).
   2. For each column in the current row:
      1. **Check if it's safe to place a queen** in the current cell:
         1. Ensure no other queen is in the same column.
         2. Ensure no other queen is on the same diagonals.
      2. If the queen can be placed safely, place it and call solveNQueens(row + 1) to place a queen in the next row.
      3. **Backtrack**: If no safe placement can be found, remove the queen from the current cell and continue with the next column.
3. **If a solution is found** (i.e., all rows are filled with queens), print the solution or store it in the result.
4. **Repeat** for all possible configurations until a solution is found or all rows are processed.

### Detailed Backtracking Algorithm:

1. **Function** isSafe(board, row, col):
   1. Checks if placing a queen in board[row][col] is safe.
   2. Ensures no other queen is placed in the same column, upper-left diagonal, or upper-right diagonal.
2. **Function** solveNQueens(board, row):
   1. Base case: If row == N, print the current board as a solution.
   2. For each column in the current row, check if placing a queen there is safe.
   3. If safe, place the queen and recursively try to place queens in the next row.
   4. If a solution is found, backtrack (remove the queen) and continue the search.
3. **Function** placeQueens(N):
   1. Initializes the board and calls solveNQueens starting from the first row.

#include <iostream> using namespace std;

#define N 4

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

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

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

cout << (placed[i] == j ? 'Q' : '.') << " ";

**}**

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

if (placed[prev] == col || placed[prev] - prev == col - row || placed[prev] + prev == col + row) { return false;

**}**

**}**

return true;

**}**

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

printSolution(placed); return;

**}**

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

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

**}**

**}**

**}**

int main() {

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

return 0;

**}**

**Input & Output :**

**Solution 1:**

**. Q . .**

**. . . Q**

**Q . . .**

**. . Q .**

**Solution 2:**

**. . Q .**

**Q . . .**

**. . . Q**

**. Q . .**

**Problem 8 : Consider a set S={5,10,12,13,15,18} and d=30. Write a program to solve the sum of subset problem.**

### Logic:

1. **Recursive Function**: For each element in the set, there are two possibilities:
   1. **Include the element** in the subset and check if the remaining sum (target sum minus the current element) can be achieved with the remaining elements.
   2. **Exclude the element** and check if the target sum can be achieved with the remaining elements.
2. **Base Case**:
   1. If the target sum becomes 0, we return True, indicating that the subset sum exists.
   2. If all elements are processed and the target sum is not 0, return False.

### Algorithm :

1. **Function** isSubsetSum(S, n, d):
   1. This function checks if there exists a subset of the first n elements in set S whose sum equals d.
2. **Recursive Backtracking**:
   1. Either include the current element or exclude it, and check if any path results in the target sum

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

int main() {

int N, target\_sum;

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

int S[N];

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

cin >> S[i];

**}**

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

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

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

bool found = false;

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

subset\_sum += S[j];

**}**

**}**

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

cout << "{ ";

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

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

**}**

**}**

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

**}**

**}**

cout << "Total subsets found: " << count << endl; return 0;

**}**

**Input & Output :**

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

**{ 5 12 13 }**

**{ 5 10 15 }**

**{ 12 18 }**

**Total subsets found: 3**

**Problem 9 : Write a program to solve the following 0/1 knapsack using dynamic programming approach profits P=(15,25,13,23), weight W=(2,6,12,9), knapsack C=20, and the number of items n=4.**

### Logic of the 0/1 Knapsack Problem using Dynamic Programming:

In the **0/1 Knapsack Problem**, we are tasked with selecting a subset of items such that the total weight of the selected items is within a given capacity and the total profit is maximized.

The key idea behind solving the problem using dynamic programming (DP) is to break the problem into smaller subproblems. Specifically, we use a DP table to store the maximum profit achievable for different combinations of items and capacities.

#### Step-by-step Logic:

**Initialization**:

* 1. We create a DP table dp where each entry dp[i][w] represents the maximum profit that can be obtained using the first i items and with a knapsack capacity of w.
  2. The table has (n + 1) rows and (C + 1) columns, where n is the number of items and C is the capacity of the knapsack.

**Recurrence Relation**:

* 1. For each item i (from 1 to n), and each possible weight capacity w (from 0 to C), we have two choices:
     1. **Exclude the item**: The maximum profit will be the same as the maximum profit without the current item, i.e., dp[i-1][w].
     2. **Include the item** (if the item can fit in the current capacity w): The maximum profit will be the sum of the current item's profit and the maximum profit that can be obtained with the remaining capacity after including the item, i.e., dp[i-1][w-W[i-1]] + P[i-1] (where W[i-1] is the weight of the item and P[i-1] is its profit).

The recurrence relation is:

dp[i][w]=max⁡(dp[i−1][w],dp[i−1][w−W[i−1]]+P[i−1])dp[i][w] = \max(dp[i-1][w], dp[i-1][w-W[i-1]] + P[i-1])dp[i][w]=max(dp[i−1][w],dp[i−1][w−W[i−1]]+P[i−1])

* 1. If the item i doesn't fit in the knapsack (W[i-1] > w), we simply exclude it and use dp[i-1][w].

**Base Case**:

* 1. When there are no items or the knapsack capacity is 0 (i == 0 or w == 0), the maximum profit is 0 because no items can be selected.

**Final Answer**:

* 1. The final answer is stored in dp[n][C], which represents the maximum profit achievable using all n items and a knapsack capacity of C.

### Dynamic Programming Algorithm for the 0/1 Knapsack Problem:

**Initialize the DP table**:

* 1. Create a DP table of size (n + 1) x (C + 1) and initialize all elements to 0.

**Fill the DP table**:

* 1. For each item i (from 1 to n), and for each capacity w (from 0 to C):
     1. If the item can be included (i.e., W[i-1] <= w), calculate the maximum profit as: dp[i][w]=max⁡(dp[i−1][w],dp[i−1][w−W[i−1]]+P[i−1])dp[i][w] = \max(dp[i-1][w], dp[i-1][w-W[i-1]] + P[i-1])dp[i][w]=max(dp[i−1][w],dp[i−1][w−W[i−1]]+P[i−1])
     2. If the item cannot be included (i.e., W[i-1] > w), use the previous value: dp[i][w]=dp[i−1][w]dp[i][w] = dp[i-1][w]dp[i][w]=dp[i−1][w]

**Retrieve the maximum profit**:

* 1. The maximum profit that can be achieved with the given items and knapsack capacity is found at dp[n][C].

### Algorithm (Step-by-step Pseudocode):

1. **Input**:
   1. P[]: Array of profits of the items.
   2. W[]: Array of weights of the items.
   3. C: Capacity of the knapsack.
   4. n: Number of items.
2. **Output**: Maximum profit that can be obtained.

#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;

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 & 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 capacity of the knapsack: 20 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**

**Problem 10 : Write a program to solve the Tower of Hanoi problem for the N disk.**

### Recursive Solution Approach:

The solution can be broken down recursively as follows:

**Base Case**:

* 1. If there is only one disk, simply move it from the source rod to the destination rod.

**Recursive Case**:

* 1. Move the top N-1 disks from the source rod to the auxiliary rod.
  2. Move the Nth disk (largest disk) directly from the source rod to the destination rod.
  3. Move the N-1 disks from the auxiliary rod to the destination rod.

This recursive approach can be represented by the following steps:

1. Move N-1 disks from the source rod to the auxiliary rod.
2. Move the Nth disk from the source rod to the destination rod.
3. Move N-1 disks from the auxiliary rod to the destination rod.

### Algorithm:

1. **Input**: The number of disks N.
2. **Output**: The sequence of moves required to solve the Tower of Hanoi problem.

### Algorithm (Recursive):

1. **Base Case**: If N == 1, simply move the disk from source to destination.
2. **Recursive Step**:
   1. Move N-1 disks from the source to auxiliary.
   2. Move the Nth disk from the source to destination.
   3. Move N-1 disks from the auxiliary to destination.

#include <iostream>

using namespace std; int main() {

int num;

char source, destination, auxiliary;

// Prompting user for input

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

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

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

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

stack[++top][0] = num; stack[top][1] = source;

stack[top][2] = destination; stack[top][3] = auxiliary;

while (top >= 0) {

int n = stack[top][0];

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

char aux\_peg = stack[top--][3];

if (n == 1) {

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

} else {

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

stack[top][3] = from\_peg;

stack[++top][0] = 1;

stack[top][1] = from\_peg; stack[top][2] = to\_peg;

stack[top][3] = aux\_peg;

stack[++top][0] = n - 1;

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

**}**

**}**

return 0;

**}**

**Input & 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**