

**Pabna University of Science and Technology**

**Information and communication Engineering**

**Lab Report**

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

**Course Code :** ICE-2202

**Submitted By: Submitted to:**

**M.Q.A Haniya Khanam Md. Anwar Hossain**

**Roll:220632 Professor**

**Session:2021-2022 Department of ICE ,PUST**

**2nd Year 2nd Semester**

**Department of ICE,PUST**

Date of submission :01.03.2025

|  |  |
| --- | --- |
| 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 marge 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 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 problem 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,the number of items n =4. |
| 10 | Write a program to solve the Tower of Hanoi problem for the N disk. |

**Index**

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

**Title:** To sort a linear array using the bubble sort algorithm.

**Theory:**

Bubble Sort is one of the simplest sorting algorithms. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated for every element in the array until the array is sorted. After each pass, the largest unsorted element is "bubbled" to its correct position. The algorithm is simple to implement but inefficient for large data sets, as its average and worst-case complexity is O(n^2).

**Algorithm:**

1. **Initialize** the array and determine its size.
2. **Outer loop**: Traverse the array multiple times. Each pass will move the largest unsorted element to its correct position.
3. **Inner loop**: For each pass, compare adjacent elements in the array. If the current element is greater than the next, swap them.
4. **Optimization**: If no swaps were made during a pass, the array is already sorted, and we can terminate early.
5. **Repeat** the process until the entire array is sorted.

**Source Code**:

#include <iostream>

using namespace std;

int main() {

int n;

// Input the size of the array

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

cin >> n;

// Declare the array

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];

}

// Bubble Sort logic

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

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

// Compare adjacent elements

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

// Swap the elements if they are in the wrong order

int temp = arr[j];

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

arr[j + 1] = temp;

}

}

}

// Output the sorted array

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

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

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

}

cout << endl;

return 0;

}

**Input:**

Enter the number of elements in the array: 5

Enter the elements of the array: 64 25 12 22 11

**Output:**

Sorted array in ascending order: 11 12 22 25 64

**Lab-2**: Write a program to find an element using a linear search

Algorithm.

**Title:** To find an element using linear search algorithm.

**Theory:**

Linear Search is a simple searching algorithm that checks every element of the array sequentially from the start to the end until the desired element (key) is found. It is the most basic search algorithm and works on both sorted and unsorted arrays.

* **Time Complexity**:
  + Best Case: O(1) (When the key is the first element).
  + Worst Case: O(n) (When the key is the last element or not present in the array).
* **Space Complexity**: O(1) (Constant space, as we only need a few extra variables).

**Algorithm:**

1. **Input** the size of the array n.
2. **Input** the elements of the array.
3. **Input** the element key to search.
4. **Loop** through the array from the first element to the last:
   * For each element, check if it matches the key.
   * If the element matches, print the index and terminate the search.
5. If the loop completes without finding the element, print a message indicating that the element was not found.

**Source Code:**

#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:**

Enter the number of elements in the array: 5

Enter the elements of the array: 10 20 30 40 50

Enter the element to search: 30

**Output:**

Element 30 found at index 2.

**Lab-3:** Write a program to sort a linear array using the marge sort

algorithm.

### ****Title****: To sort a linear array using the Merge Sort Algorithm.

### ****Theory****:

Merge Sort is a **divide-and-conquer** algorithm. It works by recursively dividing the array into smaller sub-arrays until each sub-array has only one element (which is trivially sorted). After dividing the array, Merge Sort combines (or "merges") the smaller sub-arrays back together in a sorted manner. The key idea is to divide the problem into smaller pieces and solve them independently, then combine the results.

#### **Time Complexity**:

* **Best, Worst, and Average Case**: O(nlogn)
* Merge Sort has the same time complexity regardless of the input data order. It is faster than simple algorithms like Bubble Sort and Selection Sort, especially for large datasets.

#### **Space Complexity**:

* O(n), as Merge Sort requires extra space to store the temporary sub-arrays during the merging process.

**Algorithm:**

1. **MergeSort(arr[], left, right)**:
   * If left < right:
     + Calculate the middle index: mid = (left + right) / 2.
     + Recursively sort the left half: MergeSort(arr, left, mid).
     + Recursively sort the right half: MergeSort(arr, mid + 1, right).
     + Merge the two sorted halves: Merge(arr, left, mid, right).
2. **Merge(arr[], left, mid, right)**:
   * Create two temporary arrays: leftArr[] and rightArr[].
   * Copy the elements from arr[left...mid] into leftArr[] and arr[mid+1...right] into rightArr[].
   * Merge the two sub-arrays back into the original array arr[] in sorted order.

**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 elements 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:**

Enter the elements number of array : 6

Enter the elements of array : 38 27 43 3 9 82

**Output:**

The sorted array is : 3 9 27 38 43 82

**Lab-4:** Write a program to find an element using the binary search

algorithm.

**Title:** Finding an Element Using Binary Search Algorithm.

**Theory:**

**Binary Search** is a fast and efficient algorithm for finding an element in a **sorted array**. Unlike linear search, which checks each element one by one, binary search works by dividing the search interval in half repeatedly, significantly reducing the number of comparisons.

Binary Search operates on the principle of **divide and conquer**. If the target element is smaller than the middle element of the array, the search continues in the left half of the array. If the target element is larger, the search continues in the right half. The process is repeated until the element is found or the search interval becomes empty.

**Properties of Binary Search:**

* **Time Complexity**:
  + **Best, Average, and Worst Case**: O(logn), where n is the number of elements in the array.
* **Space Complexity**:
  + **O(1)** if implemented iteratively (constant space).
  + **O(logn)** for recursive implementations due to function call stack.
* **Precondition**: The array must be sorted for binary search to work correctly.

Algorithm:

1. **Binary Search Algorithm**:
   * **Input**: A sorted array arr[] of size n and the target element key to search for.
   * **Output**: The index of the element if found, otherwise -1.
   * **Steps**:
     1. Set left = 0 and right = n - 1.
     2. While left <= right:
        + Calculate the middle index: mid = left + (right - left) / 2.
        + If arr[mid] == key, return mid (element found).
        + If arr[mid] < key, set left = mid + 1 (search in the right half).
        + If arr[mid] > key, set right = mid - 1 (search in the left half).
     3. If the element is not found, return -1.

**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 the key is present at mid

if (arr[mid] == key) {

return mid;

}

// If the key is greater than arr[mid], ignore the left half

if (arr[mid] < key) {

left = mid + 1;

}

// If the key is smaller than arr[mid], ignore the right half

else {

right = mid - 1;

}

}

// Element is not present in the array

return -1;

}

int main() {

int n, key;

// Input the size of the array

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

cin >> n;

int arr[n];

// Input the sorted elements of the array

cout << "Enter the sorted 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;

// Perform binary search

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

// Output the result

if (result != -1) {

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

} else {

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

}

return 0;

}

**Input:**

Enter the number of elements in the array: 6

Enter the sorted elements of the array: 2 5 9 12 15 18

Enter the element to search: 12

**Output:**

Element 12 found at index 3

**Lab-5:** Write a program to find a given pattern from text using the

Pattern matching algorithm.

### ****Title****: Finding a Given Pattern from Text Using Pattern Matching algorithm.

### ****Theory****:

**Pattern Matching** is the process of finding a specific sequence (pattern) of characters within a larger sequence (text). This operation is fundamental in many applications such as text search engines, bioinformatics (searching DNA sequences), and text processing tools.

In this lab, we will implement the **Naive Pattern Matching Algorithm**, which is one of the simplest methods for finding a pattern in a given text. The basic idea is to slide the pattern over the text one character at a time and compare the pattern with the substring of the text. If a match is found, the index of the starting point of the match is returned.

* **Time Complexity**: O((n - m + 1) \* m), where n is the length of the text, and m is the length of the pattern.
* **Space Complexity**: O(1), as it only requires a constant amount of extra space.

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

#### **Naive Pattern Matching Algorithm**:

1. **Input**: A text string text[] of length n and a pattern string pattern[] of length m.
2. **Output**: The index or indices where the pattern is found in the text.
3. **Steps**:
   1. Iterate through the text from i = 0 to n - m.
   2. For each position i, check if the substring starting at i matches the pattern.
      * If text[i + j] == pattern[j] for all j from 0 to m - 1, a match is found.
   3. If a match is found, return the starting index i.
   4. Repeat the process until the entire text is searched.
   5. If no match is found, return -1.

**Source Code:**

#include <iostream>

#include <string>

using namespace std;

// Naive Pattern Matching Function

int naivePatternSearch(string text, string pattern) {

int n = text.length();

int m = pattern.length();

// Loop through the text

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

int j = 0;

// Check if pattern matches the substring of text starting from index i

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

j++;

}

// If the entire pattern is found, return the starting index

if (j == m) {

return i;

}

}

// If the pattern is not found, return -1

return -1;

}

int main() {

string text, pattern;

// Input the text and pattern

cout << "Enter the text: ";

getline(cin, text);

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

getline(cin, pattern);

// Perform Naive Pattern Search

int result = naivePatternSearch(text, pattern);

// Output the result

if (result != -1) {

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

} else {

cout << "Pattern not found in the text." << endl;

}

return 0;

}

**Input:**

Enter the text: hello world, welcome to the world of programming

Enter the pattern to search: world

**Output:**

Pattern found at index: 6

**Lab-6:** Write a program to implement a queue data structure along with

typical operations.

**Title:** Implementation of Queue Data Structure in with Typical Operations.

**Theory:**

A **Queue** is a linear data structure that follows the **First In First Out (FIFO)** principle. In a queue, the element added first is the one to be removed first. It is similar to a real-world queue, like a line at a supermarket checkout where the first person in line is the first to be served.

In a queue, there are two main operations:

* **Enqueue**: Adds an element to the end of the queue.
* **Dequeue**: Removes an element from the front of the queue.

Additional operations commonly used with queues are:

* **Front**: Returns the front element of the queue without removing it.
* **IsEmpty**: Checks whether the queue is empty.
* **IsFull**: Checks whether the queue is full (relevant in fixed-size queues).
* **Size**: Returns the number of elements in the queue.

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

1. **Circular Queue Operations**:
   * **Enqueue**:
     + If the queue is full ((rear + 1) % capacity == front), display "Queue is full."
     + Otherwise, add the element at the rear position and increment rear circularly: rear = (rear + 1) % capacity.
   * **Dequeue**:
     + If the queue is empty (front == rear), display "Queue is empty."
     + Otherwise, remove the element from the front position and increment front circularly: front = (front + 1) % capacity.
   * **Front**:
     + Return the element at the front position.
   * **Size**:
     + Calculate size as (rear - front + capacity) % capacity.
   * **IsEmpty**:
     + Check if front == rear.
   * **IsFull**:
     + Check if (rear + 1) % capacity == front.

**Source Code:**

#include <iostream>

using namespace std;

class Queue {

private:

int front, rear, capacity;

int \*queue;

public:

// Constructor to initialize the queue

Queue(int size) {

capacity = size;

front = rear = 0;

queue = new int[capacity];

}

// Destructor to free memory

~Queue() {

delete[] queue;

}

// Check if the queue is empty

bool isEmpty() {

return front == rear;

}

// Check if the queue is full

bool isFull() {

return rear == capacity;

}

// Enqueue: Add an element to the queue

void enqueue(int value) {

if (isFull()) {

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

} else {

queue[rear] = value;

rear++;

cout << value << " enqueued to the queue." << endl;

}

}

// Dequeue: Remove an element from the queue

void dequeue() {

if (isEmpty()) {

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

} else {

cout << queue[front] << " dequeued from the queue." << endl;

front++;

}

}

// Front: Access the front element

int getFront() {

if (isEmpty()) {

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

return -1;

} else {

return queue[front];

}

}

// Size: Return the number of elements in the queue

int size() {

return rear - front;

}

};

int main() {

int size;

// Input the size of the queue

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

cin >> size;

Queue q(size);

// Performing queue operations

q.enqueue(10);

q.enqueue(20);

q.enqueue(30);

cout << "Front element: " << q.getFront() << endl;

q.dequeue();

cout << "Front element: " << q.getFront() << endl;

q.enqueue(40);

cout << "Queue size: " << q.size() << endl;

q.dequeue();

q.dequeue();

q.dequeue();

// Trying to dequeue from an empty queue

q.dequeue();

return 0;

}

**Input:**

Enter the size of the queue: 5

**Output:**

10 enqueued to the queue.

20 enqueued to the queue.

30 enqueued to the queue.

10 dequeued from the queue.

Front element: 20

40 enqueued to the queue.

Queue size: 2

20 dequeued from the queue.

30 dequeued from the queue.

40 dequeued from the queue.

Queue is empty!

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

#### **Title:**

#### **Solving the N-Queens Problem Using Backtracking**

#### **Theory:**

The **N-Queens problem** is a classic example of backtracking. It is defined as placing N queens on an N×N chessboard such that no two queens threaten each other. A queen can attack any other piece in the same row, column, or diagonal.

**Backtracking** is a technique used to solve problems by incrementally building candidates to the solution, and abandoning those candidates as soon as it is determined they cannot lead to a valid solution. In the N-Queens problem, backtracking explores possible configurations row by row and removes a queen from a row if placing it leads to a conflict.

1. **Row Constraint:** Only one queen can be placed in each row.
2. **Column Constraint:** No two queens can be placed in the same column.
3. **Diagonal Constraint:** No two queens can be placed on the same diagonal. Diagonals can be checked by comparing the differences between row and column indices.

**Algorithm:**

 Initialize a chessboard of size N×N, filled with zeros, representing an empty board.

 Start from the first row (row 0) and attempt to place queens in each column of the row.

 For each column, check if placing a queen results in a valid configuration (check column and both diagonals for conflicts).

 If placing the queen is valid, move to the next row and repeat the process.

 If a queen can be placed in all rows, a valid solution is found. Print the board.

 If placing a queen is not possible in any column of a row, backtrack by removing the queen from the previous row and try another column.

 Repeat the process until a solution is found or all possibilities have been exhausted.

**Source Code:**

#include <iostream>

#include <vector>

using namespace std;

void printBoard(const vector<vector<int>>& board, int n) {

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

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

if (board[i][j] == 1) {

cout << "Q "; // Print Queen's position

} else {

cout << ". "; // Empty space

}

}

cout << endl;

}

}

bool isSafe(const vector<vector<int>>& board, int row, int col, int n) {

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

if (board[i][col] == 1) {

return false;

}

}

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

if (board[i][j] == 1) {

return false;

}

}

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

if (board[i][j] == 1) {

return false;

}

}

return true;

}

bool solveNQueens(vector<vector<int>>& board, int row, int n) {

if (row == n) {

return true;

}

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

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

board[row][col] = 1;

if (solveNQueens(board, row + 1, n)) {

return true; // If placing queen leads to a solution, return true

}

board[row][col] = 0;

}

}

return false; // No safe position found in this row

}

void solveNQueens(int n) {

vector<vector<int>> board(n, vector<int>(n, 0));

if (solveNQueens(board, 0, n)) {

cout << "Solution for " << n << "-Queens Problem:" << endl;

printBoard(board, n);

} else {

cout << "No solution exists for " << n << "-Queens Problem" << endl;

}

}

int main() {

int n;

cout << "Enter the number of queens (N): ";

cin >> n;

solveNQueens(n);

return 0;

}

**Input:**

Enter the number of queens (N): 4

**Output:**

Solution for 4-Queens Problem:

Q . . .

. . Q .

. Q . .

. . . Q

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

#### **Title: Solving the Sum of Subset Problem Using Backtracking**

#### **Theory:**

The **Sum of Subset Problem** is a classic problem in computer science where, given a set SSS of positive integers and a target sum ddd, the task is to determine if there exists a subset of SSS whose sum is equal to ddd.

The problem is a special case of the **knapsack problem** and is often solved using **backtracking**. Backtracking is a method for solving problems incrementally, where partial solutions are built step by step, and as soon as a solution is deemed invalid, it is abandoned and the algorithm backtracks to try a different path.

For this problem, the goal is to explore all possible subsets of the given set SSS and check if any of them sum to the target value ddd.

##### **Backtracking for the Sum of Subset Problem:**

The backtracking approach explores each element of the set SSS and decides whether to include it in the current subset or not. At each step, the algorithm moves forward to the next element, maintaining the current sum of selected elements. If at any point the current sum equals ddd, a valid subset has been found. If the current sum exceeds ddd, that subset is discarded. If the algorithm has considered all possibilities and no solution is found, the problem has no solution.

#### **Algorithm:**

The algorithm follows the steps outlined below:

1. **Input:**
   * A set SSS of integers: S={5,10,12,13,15,18}S = \{ 5, 10, 12, 13, 15, 18 \}S={5,10,12,13,15,18}
   * A target sum d=30d = 30d=30
2. **Output:**
   * A subset of SSS whose sum equals ddd, or a message indicating no such subset exists.
3. **Steps:**
   * Start from the first element of the set and try including it in the current subset.
   * Keep track of the current sum of the subset.
   * If the current sum equals ddd, print the subset and terminate.
   * If the current sum exceeds ddd, discard this subset and backtrack.
   * Continue exploring with or without the current element, and repeat until all subsets have been checked.

**Source Code:**

#include <iostream>

#include <vector>

using namespace std;

// Function to find all subsets and check their sum

void findSubsets(const vector<int>& S, int N, int target\_sum) {

int total\_subsets = 1 << N; // Total number of subsets is 2^N

int count = 0;

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

int subset\_sum = 0;

// Generate the current subset based on the bits of mask

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

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

subset\_sum += S[j]; // Add the element to the subset sum if the j-th bit is set

}

}

// If the sum of the current subset equals the target sum, print the subset

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

cout << "{ ";

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

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

cout << S[j] << " "; // Print the element if the j-th bit is set

}

}

cout << "}\n";

count++;

}

}

// Print the total number of subsets found

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

}

int main() {

// Set S and target sum d

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

int target\_sum = 30;

// Call function to find subsets that sum up to the target sum

findSubsets(S, S.size(), target\_sum);

return 0;

}

**Input:**

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

Target sum d = 30

**Output:**

{5,10,15}

{12,18}

Total subsets found: 2

**Lab -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,the number of items n =4

#### **Title**:

Solving the 0/1 Knapsack Problem using Dynamic Programming

#### **Theory**:

The **0/1 Knapsack Problem** is an optimization problem that is commonly encountered in combinatorial optimization. It involves a set of items, each with a weight and a profit. The goal is to determine the most valuable combination of items to include in a knapsack without exceeding its capacity. The problem is called "0/1" because each item can either be included or excluded from the knapsack (i.e., the decision is binary).

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

Dynamic programming uses a bottom-up approach where we solve smaller subproblems first and build up to solve the original problem. The 2D array dp is used to store the optimal solutions of subproblems. The final solution for the full problem is stored in dp[n][C], where n is the number of items, and C is the knapsack capacity.

#### **Algorithm**:

1. **Input:**
   * Array p[] of size n containing the profit values for each item.
   * Array W[] of size n containing the weight values for each item.
   * Capacity C of the knapsack.
   * Number of items n.
2. **Create a 2D DP array dp[n+1][C+1]** where dp[i][w] represents the maximum profit achievable by using the first i items and a knapsack with capacity w.
3. **Initialization:**
   * Set all entries in dp[][] to 0. This is because if there are no items or the knapsack capacity is zero, the profit is zero.
4. **DP Recursion:**
   * For each item i (from 1 to n), and for each weight capacity w (from 0 to C):
     + If w >= W[i-1], the item can be included.

**Source Code:**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int knapsack(int C, vector<int>& W, vector<int>& p, int n) {

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

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

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

if (W[i-1] <= w) {

dp[i][w] = max(dp[i-1][w], dp[i-1][w - W[i-1]] + p[i-1]);

} else {

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

}

}

}

return dp[n][C];

}

int main() {

vector<int> p = {15, 25, 13, 23};

vector<int> W = {2, 6, 12, 9};

int C = 20;

int n = 4;

int max\_profit = knapsack(C, W, p, n);

cout << "Maximum profit that can be obtained: " << max\_profit << endl;

return 0;

}

**Input:**

Profits: {15, 25, 13, 23}

Weights: {2, 6, 12, 9}

Knapsack Capacity: 20

Number of Items: 4

**Output:**

Maximum profit that can be obtained: 38

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

#### **Title:**

Solving the Tower of Hanoi Problem Using Recursion

#### **Theory:**

The **Tower of Hanoi** is a classic problem in computer science and mathematics. The problem involves three rods and a number of disks of different sizes that can slide onto any rod. The challenge is to move the disks from one rod to another, following these specific rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack.
3. No disk may be placed on top of a smaller disk.

The goal is to move all disks from the source rod to the destination rod, using the auxiliary rod as an intermediate, while following the above rules.

The Tower of Hanoi problem is usually solved using **recursion**. The idea is to break down the problem of moving N disks into smaller subproblems.

For N disks:

1. Move the top N-1 disks from the source rod to the auxiliary rod.
2. Move the Nth (largest) disk from the source rod to the destination rod.
3. Move the 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 that solves the Tower of Hanoi problem for N disks.
3. **Recursive Function:**
   * The recursive function T(n, source, destination, auxiliary) works as follows:
     1. If n == 1, move the disk directly from the source to the destination.
     2. Otherwise:
        + Move n-1 disks from the source to the auxiliary rod.
        + Move the nth disk from the source to the destination.
        + Move n-1 disks from the auxiliary rod to the destination.
4. **Base Case:**
   * When there is only one disk, move it directly from the source rod to the destination rod.
5. **Recursion:**
   * Each recursive step reduces the problem size, solving the problem for n-1 disks and eventually reaching the base case.

**Source Code:**

#include <iostream>

using namespace std;

// Recursive function to solve the Tower of Hanoi problem

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

// Base case: If only one disk is left, move it from source to destination

if (n == 1) {

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

return;

}

// Move n-1 disks from source to auxiliary, using destination as auxiliary

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

// Move the nth disk from source to destination

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

// Move n-1 disks from auxiliary to destination, using source as auxiliary

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

}

int main() {

int n;

// Input the number of disks

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

cin >> n;

// Call the recursive function to solve the Tower of Hanoi problem

cout << "The sequence of moves are:" << endl;

towerOfHanoi(n, 'A', 'C', 'B'); // A is the source, C is the destination, B is the auxiliary rod

return 0;

}

**Input:**

Enter the number of disks: 3

**Output:**

The sequence of moves are:

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