

**Department of**

**Information & Communication Engineering**

**LAB REPORT**

**ICE-2202**

**Data Structure & Algorithm Sessional**

**Submitted By:**

**SAMIUL ISLAM TONMOY**

**Roll: 2206042**

**Session 2021-22**

**Dept. of Information and**

**Communication Engineering**

**Submitted To :**

**Dr. Md. Anwar Hossain**

**Professor**

**Dept. of Information and Communication Engineering**



**Laboratory Problem Index**

**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 |

**Lab Problem No:** 01

**Title:** Sorting a Linear Array Using Bubble Sort Algorithm.

**Theory:** Sorting is a fundamental operation in computer science that involves arranging elements in a specified order. Bubble Sort is a simple comparison-based sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until the list is sorted. The algorithm has a time complexity of O(n²) in the worst and average cases, making it inefficient for large datasets. However, it is easy to implement and useful for educational purposes.

**Algorithm:**

1. Start by comparing the first two adjacent elements in the array.
2. If the first element is greater than the second, swap them.
3. Move to the next adjacent pair and repeat step 2.
4. Continue this process until the last element of the array.
5. Repeat the entire process for (n-1) passes, where n is the number of elements in the array.
6. If no swaps are made in a pass, the array is already sorted, and the algorithm terminates early.
7. The sorted array is obtained after all required passes.

**Implementation of Source code(C++):**

#include <iostream>

using namespace std;

int main() {

    int n;

    //input the size

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

    cin >> n;

    int arr[n];

    //input the elements

    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++) {

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

                int temp = arr[j];

                arr[j] = arr[j + 1];

                arr[j + 1] = temp;

            }

        }

    }

    //sorted array

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

    for (int i = 0; i < n; i++) {

        cout << arr[i] << " ";

    }

    return 0;

}

**Input:**

Enter the number of elements in the array: 4

Enter the elements of the array: 5 7 2 4

**Output:**

Sorted array in ascending order: 2 4 5 7

**Lab Problem No:** 02

**Title:** Finding an Element Using Linear Search Algorithm

**Theory:** Linear Search, also known as sequential search, is a straightforward algorithm used to find a specific element within a list or array. The algorithm works by iterating through each element in the data structure, starting from the first element and moving sequentially to the last, comparing each element with the target value. If a match is found, the search is successful, and the index of the matching element is returned. If the algorithm reaches the end of the list without finding the target, it concludes that the element is not present. This method is simple to implement and does not require the data to be sorted. However, its time complexity is O(n), making it less efficient for large datasets compared to more advanced search algorithms like binary search.

**Algorithm:**

1. **Start**: Initialize the search by setting a counter (index) to the first element of the array.
2. **Input**: Obtain the target value to be searched from the user.
3. **Iteration**: While the counter is less than the total number of elements in the array:
   * Compare the current array element with the target value.
   * If they are equal, return the current index and terminate the algorithm.
   * If they are not equal, increment the counter by one and proceed to the next element.
4. **Not Found**: If the end of the array is reached without finding the target, return a message indicating that the element is not present in the array.
5. **End**: Terminate the algorithm.

**Implementation of Source code (C++):**

#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: 4

Enter the elements of the array: 5 7 8 9

Enter the element to search: 9

**Output:**

Element 9 found at index 3.

**Lab Problem No:** 03

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

**Theory:** Merge Sort is a comparison-based sorting algorithm that employs the divide-and-conquer paradigm. It recursively divides the unsorted array into two halves until each sub-array contains a single element. Then, it merges these sub-arrays in a manner that results in a sorted array. This process ensures that the entire array becomes sorted upon completion. Merge Sort is known for its stable sorting nature and guarantees a time complexity of O(n log n) in the worst, average, and best cases, making it efficient for large datasets. However, it requires additional space proportional to the size of the input array, which is a consideration in memory-constrained environments.

**Algorithm:**

1. **Divide**: If the array has more than one element, split it into two approximately equal halves.
2. **Conquer**: Recursively apply the Merge Sort algorithm to both halves to sort them.
3. **Combine**: Merge the two sorted halves into a single sorted array by comparing elements from each half and arranging them in order.

**Implementation of Source code(C++):**

#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:**

Enter the elemnets number of array : 4

Enter the elements of array : 8 4 3 6

**Output:**

The sorted array is : 3 4 6 8

**Lab Problem No:** 04

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

**Theory:** Binary Search is an efficient algorithm for locating a target value within a sorted array. It operates on the divide-and-conquer principle, repeatedly dividing the search interval in half. By comparing the target value to the middle element of the array, the algorithm determines whether to continue the search in the lower or upper half, effectively reducing the search space by half with each iteration. This method significantly improves search efficiency, achieving a time complexity of O(log n), where n is the number of elements in the array. Binary Search requires that the array be sorted prior to execution; applying it to an unsorted array yields incorrect results. Its efficiency and straightforward implementation make it a fundamental technique in computer science for search operations.

**Algorithm:**

1. **Initialization**: Set two pointers, low to the first index (0) and high to the last index (n - 1) of the array.
2. **Iteration**: While low is less than or equal to high:
   * Calculate the middle index: mid = low + (high - low) // 2.
   * Compare the middle element arr[mid] with the target value x:
     + If arr[mid] equals x, return mid (the index of the target).
     + If arr[mid] is less than x, set low = mid + 1 (continue search in the upper half).
     + If arr[mid] is greater than x, set high = mid - 1 (continue search in the lower half).
3. **Termination**: If low exceeds high, the target value is not present in the array; return -1 to indicate unsuccessful search.

**Implementation of Source code (C++):**

#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:**

Enter the number of elements: 4

Enter 4 sorted elements: 3 5 6 9

Enter the number to search: 6

**Output:**

Target 6 found at position 3

**Lab Problem No:** 05

**Title:** Finding a Given Pattern in Text Using Pattern Matching Algorithms

**Theory:** Pattern matching is a fundamental operation in computer science, involving the search for occurrences of a specific sequence of characters, known as a pattern, within a larger text. Efficient pattern matching is crucial in various applications, including text editing, search engines, DNA sequencing, and network security. Several algorithms have been developed to perform pattern matching, each with its own advantages and use cases. Two prominent algorithms are:

1. **Knuth-Morris-Pratt (KMP) Algorithm**: Developed by Donald Knuth, James Morris, and Vaughan Pratt, the KMP algorithm preprocesses the pattern to create a partial match table (also known as the "failure function"), which indicates the longest proper prefix of the pattern that is also a suffix. This allows the algorithm to avoid redundant comparisons, achieving a time complexity of O(n + m), where n is the length of the text and m is the length of the pattern.
2. **Rabin-Karp Algorithm**: Introduced by Michael Rabin and Richard Karp, this algorithm uses hashing to find any one of a set of pattern strings in a text. It computes a hash value for the pattern and each substring of the text of the same length as the pattern. If the hash values match, it performs a direct comparison to confirm the match. The average-case time complexity is O(n + m), but the worst-case complexity can degrade to O(nm) due to hash collisions.

**Algorithm:**

***Knuth-Morris-Pratt (KMP) Algorithm*:**

1. **Preprocessing Phase:**
   * Construct the partial match table for the pattern.
   * Initialize the table with the first value as -1 and the rest as 0.
   * Use two pointers to traverse the pattern and fill the table based on the longest proper prefix which is also a suffix.
2. **Searching Phase:**
   * Initialize pointers for the text and pattern.
   * While the text pointer is within the bounds of the text:
     + If the current characters of the text and pattern match, increment both pointers.
     + If the pattern pointer reaches the end of the pattern, a match is found.
     + If a mismatch occurs and the pattern pointer is not at the beginning, use the partial match table to shift the pattern pointer.
     + If a mismatch occurs and the pattern pointer is at the beginning, move the text pointer to the next position.

***Rabin-Karp Algorithm*:**

1. **Preprocessing Phase:**
   * Compute the hash value of the pattern.
   * Compute the hash value of the first substring of the text with the same length as the pattern**.**
2. **Searching Phase:**
   * Slide the pattern over the text one character at a time.
   * For each position, compare the hash value of the current substring of the text with the hash value of the pattern**.**
   * If the hash values match, perform a direct character-by-character comparison to confirm the match**.**
   * If a mismatch occurs, compute the hash value for the next substring by removing the leadingcharacter and adding the trailing character (rolling hash).

**Implementation of source code(C++):**

#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:**

Enter the text: hello world

Enter the pattern: wor

**Output :**

Pattern found at index 6

**Lab Problem No:** 03

**Title:** Implementation of a Queue Data Structure and Its Typical Operations.

**Theory:**

A queue is a fundamental linear data structure in computer science that operates on the First-In-First-Out (FIFO) principle. This means that the first element added to the queue will be the first one to be removed. Queues are analogous to lines in real life, such as a line of customers waiting for service, where the first customer in line is the first to be served.

Common applications of queues include:

* Resource scheduling: Managing tasks in operating systems, such as print spooling or process scheduling.
* Data buffering: Storing data temporarily while it's being transferred between processes or devices, like in IO operations.
* Handling asynchronous data: Managing data from different sources that arrive at different rates, such as keyboard strokes or network packets.

Basic operations of a queue:

1. Enqueue: Adding an element to the rear (end) of the queue.
2. Dequeue: Removing an element from the front (beginning) of the queue.
3. Peek/Front: Retrieving the element at the front of the queue without removing it.
4. IsEmpty: Checking whether the queue is empty.
5. IsFull: (For bounded queues) Checking whether the queue has reached its capacity.

Queues can be implemented using various data structures, including arrays and linked lists. In an array-based implementation, the queue has a fixed size, and two pointers (or indices) are used to track the front and rear positions. In a linked list implementation, the queue can dynamically adjust its size, with nodes linked together, each containing data and a reference to the next node.

**Algorithm:**

*Implementing a Queue Using an Array:*

1. **Initialize the Queue:**
   * Define the maximum size of the queue.
   * Create an array of the defined size to hold the queue elements.
   * Initialize two integer variables, front and rear, to -1 to represent an empty queue.
2. **Enqueue Operation:**
   * **Check if the queue is full:**
     + If rear is equal to the maximum size minus one (rear == SIZE - 1), the queue is full, and insertion is not possible.
   * **Insert the new element:**
     + If the queue is not full:
       - If the queue is empty (front == -1), set front to 0.
       - Increment rear by 1.
       - Add the new element at the position indicated by rear in the array.
3. **Dequeue Operation:**
   * **Check if the queue is empty:**
     + If front is equal to -1 or front is greater than rear, the queue is empty, and deletion is not possible.
   * **Remove the front element:**
     + If the queue is not empty:
       - Retrieve the element at the position indicated by front in the array.
       - Increment front by 1.
       - If front becomes greater than rear, reset both front and rear to -1 to indicate that the queue is now empty.
4. **Peek/Front Operation:**
   * **Check if the queue is empty:**
     + If front is equal to -1 or front is greater than rear, the queue is empty, and there is no front element to retrieve.
   * **Retrieve the front element:**
     + If the queue is not empty, return the element at the position indicated by front in the array.
5. **IsEmpty Operation:**
   * Return true if front is equal to -1 or front is greater than rear; otherwise, return false.
6. **IsFull Operation:**
   * Return true if rear is equal to the maximum size minus one (rear == SIZE - 1); otherwise, return false.

*Implementing a Queue Using a Linked List:*

1. **Define the Node Structure:**
   * Each node contains two parts:
     + data: To store the value of the element.
     + next: A pointer/reference to the next node in the queue.
2. **Initialize the Queue:**
   * Maintain two pointers, front and rear, both initially set to null, representing an empty queue.
3. **Enqueue Operation:**
   * **Create a new node:**
     + Allocate memory for a new node and assign the given value to its data field.
     + Set its next pointer to null.
   * **Insert the new node:**
     + If the queue is empty (front == null):
       - Set both front and rear to point to the new node.
     + If the queue is not empty:
       - Set the next pointer of the current rear node to point to the new node.
       - Update the rear pointer to point to the new node.
4. **Dequeue Operation:**
   * **Check if the queue is empty:**
     + If front is null, the queue is empty, and deletion is not possible.

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

#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;

}

**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...

**Lab Problem No: 07**

**Title:** Solving the N-Queens Problem Using Backtracking

**Theory:**

The N-Queens problem is a classic combinatorial puzzle that involves placing N chess queens on an N×N chessboard such that no two queens threaten each other. This means ensuring that no two queens share the same row, column, or diagonal. The problem serves as an excellent example to illustrate the backtracking algorithm—a systematic method for exploring all potential configurations in search of a solution. Backtracking incrementally builds candidates for the solution and abandons ("backtracks") a candidate as soon as it determines that the candidate cannot possibly lead to a valid solution. This approach is particularly useful for constraint satisfaction problems like the N-Queens puzzle.

**Algorithm:**

1. Initialize the Board:
   * Create an N×N chessboard initialized with zeros, where zeros represent empty squares.
2. Define the Backtracking Function:
   * Develop a recursive function that attempts to place queens column by column.
3. Base Case:
   * If all queens are placed (i.e., the current column index equals N), print the board configuration as a solution.
4. Recursive Case:
   * For the current column, iterate over each row to attempt placing a queen:
     + Check Safety: Determine if placing a queen at the current row and column is safe by ensuring no other queens threaten this position.
     + Place the Queen: If the position is safe, mark the square with a 1 (indicating a queen is placed).
     + Recur to Next Column: Recursively attempt to place queens in the subsequent columns.
     + Backtrack if Needed: If placing a queen in the current configuration doesn't lead to a solution, remove the queen (backtrack) and continue with the next row.
5. Safety Check Function:
   * Implement a helper function to verify that placing a queen at a given position doesn't result in any conflicts with already placed queens. This involves checking:
     + The same row on the left side.
     + The upper diagonal on the left side.
     + The lower diagonal on the left side.

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

#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;

}

**Output**

**Solution 1:**

. Q . .

. . . Q

Q . . .

. . Q .

**Solution 2:**

. . Q .

Q . . .

. . . Q

. Q .

**Lab Problem No: 8**

**Title**: Sum of Subset Problem

**Theory:**

he Sum of Subset problem is a classic combinatorial problem in which we determine whether there exists a subset of a given set whose sum equals a specified target value. This problem can be solved using techniques such as:

* Backtracking: Recursively exploring subsets while pruning unnecessary computations.
* Dynamic Programming: Using a tabular approach to build possible subset sums efficiently.
* Bitmasking/Brute Force: Generating all possible subsets and checking their sum.

In this case, we are given a set S = {5, 10, 12, 13, 15, 18} and a target sum d = 30. Our goal is to find subsets whose sum equals 30.

**Algorithm: (Using Backtracking)**

1. **Sort the set** in non-decreasing order.
2. **Initialize a recursive function** to explore possible subsets.
3. **At each step:**
   * Include the current element in the subset and check if the sum is achieved.
   * If not, move forward and explore further elements.
   * If a valid subset is found, print/store the result.
4. **Backtrack** when the sum exceeds the target or all elements are considered.

**Implementation of Source Code (C++)**

#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;

}

**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

**Lab Problem No: 9**

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

**Theory:**

The core idea behind the dynamic programming solution is to build a table (matrix) dp where dp[i][w] represents the maximum value that can be obtained with a knapsack of capacity w using items up to index i.

We fill the table using the following recurrence relation:

* If the weight of the current item W[i-1] is greater than the current capacity w, then dp[i][w] = dp[i-1][w] (we cannot include the item).
* Otherwise, dp[i][w] = max(dp[i-1][w], P[i-1] + dp[i-1][w - W[i-1]]) (we choose the maximum between not including the item and including it).

**Given Data**

* Profits (P): (15, 25, 13, 23)
* Weights (W): (2, 6, 12, 9)
* Knapsack Capacity (C): 20
* Number of items (n): 4

**Algorithm:**

**Step 1: Define the Problem**

Given:

* n items, each with a weight w[i] and a value v[i]
* A knapsack with a maximum weight capacity W
* You can either **include** or **exclude** each item (hence 0/1)

**Step 2: Define DP State**

Let dp[i][j] represent the **maximum value** that can be obtained using the first i items with a weight limit of j.

**Step 3: Recursive Relation**

For each item i, we have two choices:

1. **Exclude** the item → Value remains the same as dp[i-1][j]
2. **Include** the item (if weight allows) → Add the item's value v[i-1] and reduce the capacity accordingly.

**Step 4: Initialize DP Table**

* dp[0][j] = 0 for all j (no items → zero value)
* dp[i][0] = 0 for all i (zero capacity → zero value)

**Step 5: Compute DP Table**

Iterate through each item and weight capacity to fill dp[][].

**Step 6: Extract Solution**

The answer is stored in dp[n][W] (maximum value with n items and weight W).

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

#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;

}

**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:

tem 4 (Profit: 23, Weight: 9)

Item 2 (Profit: 25, Weight: 6)

Item 1 (Profit: 15, Weight: 2)

Maximum profit in the knapsack: 63

**Lab Problem No: 10**

**Title:** Solving the Tower of Hanoi Problem for n Disks

**Theory:** The Tower of Hanoi is a mathematical puzzle that involves three rods and n disks of different sizes. The goal is to move all disks from the source rod to the destination rod while following these constraints:

1. Only one disk can be moved at a time.
2. A disk can only be placed on top of a larger disk or an empty rod.
3. Only the top disk of any rod can be moved.

The problem is a classic example of recursion, where the solution to a larger problem depends on solving smaller instances of the same problem. The number of moves required to solve the Tower of Hanoi with n disks is given by: which simplifies to .

**Algorithm:**

1. Move 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 thse auxiliary rod to the destination rod.

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

#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;

}

Output : Enter the number of disk

s: 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