## 

PABNA UNIVERSITY OF SCIENCE AND TECHNOLOGY

Faculty Of Engineering & Technology

Department Of

Information And Communication Engineering

**Submitted To :**

Md. Anwar Hossain

Professor

Department of Information & Communication Engineering

Pabna University of Science and Technology

Course Code : ICE - 2202

Course Title : Data Structure and Algorithoms Sessional

LAB REPORT

**Submitted By :**

Name : MD MAHAFUG AHMED

Roll No : 220628

Session : 2021-2022

2nd Year 2nd Semester

Department of ICE,PUST

Submission Date : 01-03-2025

**Index**

|  |  |
| --- | --- |
| **Sl.** | **Problem Statement** |
| **1.** | |  | | --- | | Write a program to sort a linear array using the bubble sort algorithm. | |
| **2.** | |  | | --- | | Write a program to find an element using a linear search algorithm. | |
| **3.** | |  | | --- | | Write a program to sort a linear array using the merge sort algorithm. | |
| **4.** | |  | | --- | | Write a program to find an element using the binary search algorithm. | |
| **5.** | |  | | --- | | Write a program to find a given pattern from text using the pattern matching algorithm. | |
| **6.** | |  | | --- | | Write a program to implement a queue data structure along with its typical operations. | |
| **7.** | |  | | --- | | Write a program to solve **n** queen's problem using backtracking. | |
| **8.** | |  | | --- | | Consider a set **S = {5,10,12,13,15,18}** and **d = 30**. Write a program to solve the sum of subset 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**. | |
| **10** | |  | | --- | | Write a program to solve the **Tower of Hanoi** problem for the **N** disk. | |

**Problem No : 01**

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

**Theory:**

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

**Time Complexity:**

* Best Case: O(n) (When the array is already sorted)
* Worst Case: O(n²) (When the array is sorted in reverse order)
* Average Case: O(n²)

**Space Complexity:** O(1) (In-place sorting algorithm)

**Algorithm:**

1. Start by iterating through the array.
2. Compare each pair of adjacent elements.
3. Swap them if they are in the wrong order.
4. Repeat the process for the remaining elements.
5. Continue this until no more swaps are needed.

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

#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]) {

                // Swap elements

                int temp = arr[j];

                arr[j] = arr[j + 1];

                arr[j + 1] = temp;

                swapped = true;

            }

        }

        // If no swaps occurred, the array is already sorted

        if (!swapped) break;

    }

}

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 elements: ";

    cin >> n;

    int arr[n];

    cout << "Enter " << n << " elements: ";

    for (int i = 0; i < n; i++) {

        cin >> arr[i];

    }

    cout << "Original array: ";

    printArray(arr, n);

    bubbleSort(arr, n);

    cout << "Sorted array: ";

    printArray(arr, n);

    return 0;

}

**Sample Input and Output:**

**Input :**

Enter the number of elements: 5

Enter 5 elements: 64 34 25 12 22

**Output :**

Original array: 64 34 25 12 22

Sorted array: 12 22 25 34 64

**Problem No : 02**

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

**Theory :**

Linear search is a simple searching algorithm used to find an element in a list or an array. It works by sequentially checking each element of the list until the desired element is found or the list ends.

* **Best Case:** The element is found at the first position, requiring only one comparison (**O(1)**).
* **Worst Case:** The element is at the last position or not present in the array, requiring **O(n)** comparisons.
* **Average Case Complexity:** **O(n)**

Linear search is inefficient for large datasets but is simple to implement and does not require sorted data.

**Algorithm:**

1. Start from the first element of the array.
2. Compare the target element with the current element.
3. If a match is found, return the position.
4. If the end of the array is reached and no match is found, return -1 to indicate the element is not present.

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

#include<iostream>

using namespace std;

// Function to perform linear search

int linearSearch(int arr[], int size, int key) {

    for (int i = 0; i < size; i++) {

        if (arr[i] == key) {

            return i; // Return index of found element

        }

    }

    return -1; // Return -1 if element is not found

}

int main() {

    int n, key;

    // Taking array size input

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

    cin >> n;

    int arr[n];

    // Taking array elements input

    cout << "Enter " << n << " elements: ";

    for (int i = 0; i < n; i++) {

        cin >> arr[i];

    }

    // Taking the search key input

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

    cin >> key;

    // Calling linear search function

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

    // Display result

    if (result != -1) {

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

    } else {

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

    }

    return 0;

}

**Sample Input and Output:**

**Input :**

Enter the number of elements in the array: 5

Enter 5 elements: 10 20 30 40 50

Enter the element to search: 30

**Output :**

Element found at index 2

**Problem No : 03**

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

**Theory :**

Merge Sort is a divide-and-conquer sorting algorithm that splits an array into smaller subarrays, sorts them, and then merges them back together. It works by recursively dividing the array into two halves until each half has a single element, then merging them back in a sorted manner. Merge Sort has a time complexity of **O(n log n)** in all cases, making it more efficient than algorithms like Bubble Sort or Insertion Sort for large datasets.

**Algorithm:**

1. Define a function mergeSort() that takes an array and its start and end indices as parameters.
2. If the start index is less than the end index:
   * Find the middle index.
   * Recursively call mergeSort() for the left and right halves of the array.
   * Merge the sorted halves using a helper function merge().
3. The merge() function:
   * Creates temporary arrays for the left and right halves.
   * Compares elements from both halves and arranges them in sorted order.
   * Copies the sorted elements back to the original array.

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

#include <iostream>

using namespace std;

void merge(int arr[], int left, int mid, int right) {

    int n1 = mid - left + 1;

    int n2 = right - mid;

    int 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) {

        if (leftArr[i] <= rightArr[j]) {

            arr[k] = leftArr[i];

            i++;

        } else {

            arr[k] = rightArr[j];

            j++;

        }

        k++;

    }

    while (i < n1) {

        arr[k] = leftArr[i];

        i++; k++;

    }

    while (j < n2) {

        arr[k] = rightArr[j];

        j++; k++;

    }

}

void mergeSort(int arr[], int left, int right) {

    if (left < right) {

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

        mergeSort(arr, left, mid);

        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);

    }

}

int main() {

    int n;

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

    cin >> n;

    int arr[n];

    cout << "Enter " << n << " elements: ";

    for (int i = 0; i < n; i++)

        cin >> arr[i];

    cout << "Original array: ";

    for (int i = 0; i < n; i++)

        cout << arr[i] << " ";

    cout << endl;

    mergeSort(arr, 0, n - 1);

    cout << "Sorted array: ";

    for (int i = 0; i < n; i++)

        cout << arr[i] << " ";

    cout << endl;

    return 0;

}

**Sample Input and Output:**

**Input :**

Enter the number of elements: 5

Enter 5 elements: 12 9 18 3 25

**Output :**

Original array: 12 9 18 3 25

Sorted array: 3 9 12 18 25

**Problem No : 04**

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

**Theory :**

Binary Search is an efficient searching algorithm that works on sorted arrays. It follows the divide-and-conquer strategy by repeatedly dividing the search space in half until the target element is found or the search space is empty.

* **Time Complexity:**
  + **Best case:** O(1) (if the middle element is the target)
  + **Average & Worst case:** O(log *n* )
* **Space Complexity:** O(1) (for iterative approach)

**Working Principle:**

1. Start with two pointers, low (beginning of the array) and high (end of the array).
2. Calculate the middle index: mid=low+high/2
3. Compare the middle element with the target:
   * If it matches, return the index.
   * If it's smaller, search in the right half (low = mid + 1).
   * If it's larger, search in the left half (high = mid - 1).
4. Repeat until low > high (element not found).

**Algorithm:**

1. Start with low = 0 and high = n - 1 (where n is the size of the array).
2. Repeat until low is less than or equal to high:

* Compute mid = (low + high) / 2.
* If arr[mid] == key, return mid (element found).
* If arr[mid] < key, set low = mid + 1 (search in the right half).
* If arr[mid] > key, set high = mid - 1 (search in the left half).

**3**.If the loop ends, return -1 (element not found).

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

#include <iostream>

using namespace std;

int binarySearch(int arr[], int size, int key) {

    int low = 0, high = size - 1;

    while (low <= high) {

        int mid = (low + high) / 2; // Calculate middle index

        if (arr[mid] == key)

            return mid; // Element found

        if (arr[mid] < key)

            low = mid + 1; // Search in right half

        else

            high = mid - 1; // Search in left half

    }

    return -1; // Element not found

}

int main() {

    int n, key;

    // Taking array input

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

    cin >> n;

    int arr[n];

    cout << "Enter " << n << " sorted elements: ";

    for (int i = 0; i < n; i++) {

        cin >> arr[i];

    }

    // Taking the element to search

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

    cin >> key;

    // Calling binary search function

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

    if (result != -1)

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

    else

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

    return 0;

}

**Sample Input and Output:**

**Input :**

Enter the number of elements in sorted array: 5

Enter 5 sorted elements: 10 20 30 40 50

Enter the element to search: 30

**Output :**

Element found at index: 2

**Problem 5 : Pattern Matching Algorithm**

**Problem Title : Finding a Given Pattern from text Using the Pattern Matching Algorithm**

**Theory :**

Pattern matching is the process of finding a specific sequence of characters (pattern) within a larger string (text). It is widely used in text processing, search engines, and computational biology. There are various pattern-matching algorithms, such as the Naive Algorithm, Knuth-Morris-Pratt (KMP) Algorithm, and Boyer-Moore Algorithm. In this lab, we implement the **Naive Pattern Matching Algorithm**, which is simple and efficient for small inputs.

The **Naive Algorithm** works by sliding the pattern over the text one position at a time and checking if the pattern matches the substring in the text. If a match is found, the starting index is recorded.

**Algorithm:**

1. Take input: a text string and a pattern string.
2. Determine the lengths of the text (N) and pattern (M).
3. Slide the pattern over the text one character at a time:
   * Compare the pattern with the current substring of the text.
   * If all characters match, record the index.
4. Repeat the process until the entire text is scanned.
5. Print the positions where the pattern occurs.

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

#include <iostream>

using namespace std;

void naivePatternMatching(char text[], char pattern[]) {

    int N = 0, M = 0;

    // Find lengths of text and pattern manually

    while (text[N] != '\0') N++;

    while (pattern[M] != '\0') M++;

    // Iterate over text to check for pattern

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

        int j;

        for (j = 0; j < M; j++) {

            if (text[i + j] != pattern[j]) {

                break;

            }

        }

        // If the pattern is found

        if (j == M) {

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

        }

    }

}

int main() {

    char text[100], pattern[50];

    cout << "Enter the text: ";

    cin.getline(text, 100);

    cout << "Enter the pattern: ";

    cin.getline(pattern, 50);

    naivePatternMatching(text, pattern);

    return 0;

}

**Sample Input and Output:**

**Input :**

Enter the text: mahafuj

Enter the pattern: ha

**Output :**

Pattern found at index 2

**Problem 6 : Queue Implementation**

**Problem Title : Implementing a Queue Data Structure along with it’s Typical Operations.**

**Theory :**

A queue is a linear data structure that follows the **FIFO (First In, First Out)** principle, meaning elements are added at the rear (enqueue) and removed from the front (dequeue). It can be implemented using arrays or linked lists. A queue supports the following operations:

1. **Enqueue:** Adds an element to the rear of the queue.
2. **Dequeue:** Removes an element from the front of the queue.
3. **Front (Peek):** Returns the front element without removing it.
4. **IsEmpty:** Checks if the queue is empty.
5. **IsFull:** Checks if the queue is full (for array-based implementation).
6. **Display:** Shows the current elements in the queue.

**Algorithm:**

### 1. Enqueue Operation:

1. Check if the queue is full.
2. If not, increment the rear index.
3. Insert the element at the rear index.

### 2. Dequeue Operation:

1. Check if the queue is empty.
2. If not, remove the element from the front.
3. Increment the front index.

### 3. Peek Operation:

1. Check if the queue is empty.
2. If not, return the front element.

### 4. IsEmpty Operation:

1. Check if the front index is greater than the rear index.

### 5. IsFull Operation:

1. Check if the rear index has reached the maximum capacity of the queue.

### 6. Display Operation:

1. Print all elements from the front to the rear.

**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;

}

**Sample Input and Output:**

**Output :**

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 1

Enter the value to enqueue: 5

5 enqueued to the queue.

Current front: 0, rear: 0

Current queue: 5

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 1

Enter the value to enqueue: 6

6 enqueued to the queue.

Current front: 0, rear: 1

Current queue: 5 6

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 2

Dequeued value: 5

Current front: 1, rear: 1

Current queue: 6

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 1

Enter the value to enqueue: 7

7 enqueued to the queue.

Current front: 1, rear: 2

Current queue: 6 7

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 3

Queue elements are: 6 7

Current queue: 6 7

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 4

Exiting...

**Problem 7 : N-Queen’s Problem**

**Problem Title : Solving N-Queen’s problem using Backtracking.**

**Theory :**

The N-Queens problem is a classic combinatorial problem that involves placing N queens on an N × N chessboard such that no two queens threaten each other. This means that no two queens can be placed in the same row, column, or diagonal.

The problem can be solved using **backtracking**, a technique that incrementally builds a solution and abandons a path (backtracks) as soon as it determines that the current path cannot possibly lead to a valid solution.

The backtracking approach follows these steps:

1. Place a queen in the first available row.
2. Move to the next row and try placing a queen in a column that does not conflict with previously placed queens.
3. If a conflict arises, backtrack to the previous row and move the queen to the next valid column.
4. Repeat steps 2 and 3 until all queens are placed successfully or all options are exhausted.

**Algorithm:**

1. Start placing queens row by row.
2. For each row, check if placing a queen in a column is safe.
3. If it is safe, place the queen and move to the next row.
4. If placing a queen leads to a conflict, remove it (backtrack) and try the next column.
5. If all queens are placed successfully, print the solution.
6. If no valid placement exists, return to the previous row and adjust the queen's position.

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

#include <iostream>

using namespace std;

bool isSafe(int\*\* board, int row, int col, int n) {

    for (int i = 0; i < row; i++)

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

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

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

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

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

    return true;

}

bool solveNQueens(int\*\* board, 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;

            board[row][col] = 0;

        }

    }

    return false;

}

void printSolution(int\*\* board, int n) {

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < n; j++) {

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

        }

        cout << endl;

    }

}

void solve() {

    int n;

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

    cin >> n;

    int\*\* board = new int\*[n];

    for (int i = 0; i < n; i++)

        board[i] = new int[n]{0};

    if (solveNQueens(board, 0, n))

        printSolution(board, n);

    else

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

    for (int i = 0; i < n; i++)

        delete[] board[i];

    delete[] board;

}

int main() {

    solve();

    return 0;

}

**Sample Input and Output:**

**Input:**

Enter the number of queens: 8

**Output:**

Q - - - - - - -

- - - - Q - - -

- - - - - - - Q

- - - - - Q - -

- - Q - - - - -

- - - - - - Q -

- Q - - - - - -

- - - Q - - - -

**Problem 8 :** **Sum of Subset Problem**

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

**Theory :**

The **Sum of Subset Problem** is a decision problem in computer science and mathematics that asks whether there exists a subset of a given set of integers whose sum is equal to a specified value.

In this problem, you are given a set of integers and a target sum d. The task is to find out if there is a subset of the set whose sum equals d. This can be solved using dynamic programming.

#### **Problem Definition:**

Given a set S = {5, 10, 12, 13, 15, 18} and a target sum d = 30, the task is to determine whether there is a subset of S that sums up to d.

The **dynamic programming** approach works by constructing a table dp where each cell dp[i][j] represents whether a sum j can be formed using the first i elements of the set.

**Algorithm:**

1. Let the given set be S = {s1, s2, ..., sn} and the target sum be d.
2. Create a 2D table dp[i][j], where i represents the number of elements considered so far, and j represents the possible sums.
3. Initialize dp[0][0] = true, which means sum 0 can be formed with 0 elements.
4. For each element in the set, and for each sum from 0 to d:
   * If the sum j can be achieved either by excluding the current element or by including it, set dp[i][j] = true.
   * If the current sum j is not achievable, set dp[i][j] = false.
5. The answer is dp[n][d], which tells whether the sum d can be formed with all elements in the set.

**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;

}

**Sample Input and Output:**

**Input :**

Enter the number of elements: 6

Enter the elements: 5 10 12 13 15 18

Enter the target sum: 30

**Output:**

{ 5 12 13 }

{ 5 10 15 }

{ 12 18 }

Total subsets found: 3

**Problem 9: 0/1 Knapsack Problem**

**Problem Title :** Write a program to solve the following 0/1 Knapsack problem using dynamic programming. Given:

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

**Theory :**

The **0/1 Knapsack Problem** is a classical problem in optimization. The goal is to select a subset of items from a set, such that their total weight does not exceed the capacity of the knapsack, while maximizing the total profit.

* **Input:**
  + A list of **profits** (P[i]) corresponding to each item.
  + A list of **weights** (W[i]) corresponding to each item.
  + **Capacity (C)** of the knapsack.
  + **n** items.
* **Objective:**
  + Maximize the total profit subject to the constraint that the total weight of selected items does not exceed the capacity.

**Dynamic Programming Approach:**

1. **State Representation:**
   * Let dp[i][w] represent the maximum profit that can be achieved with the first i items and a knapsack capacity w.
   * If the weight of the i-th item is greater than the capacity w, then it cannot be included in the knapsack.
2. **Recurrence Relation:**
   * If item i is excluded: dp[i][w] = dp[i-1][w]
   * If item i is included: dp[i][w] = P[i-1] + dp[i-1][w-W[i-1]]

The relation is:

cpp

CopyEdit

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

1. **Base Case:**
   * dp[0][w] = 0 for all w (no items lead to zero profit).
   * dp[i][0] = 0 for all i (knapsack capacity zero means no items can be chosen).

**Algorithm:**

#### **Step 1: Input Parsing**

1. **Input the number of items, n.**
2. **Input the profits** of the n items into an array profits[].
3. **Input the weights** of the n items into an array weights[].
4. **Input the knapsack's capacity, C.**

#### **Step 2: Dynamic Programming Table Initialization**

1. **Create a 2D table dp[][]** where:
   * dp[i][w] represents the maximum profit achievable with the first i items and a knapsack capacity w.
   * The table dimensions will be (n+1) x (C+1) to account for n items and capacities from 0 to C.
2. **Initialize the first row (dp[0][w]) and the first column (dp[i][0])** to 0:
   * dp[0][w] = 0 for all w: When there are no items, the maximum profit is zero.
   * dp[i][0] = 0 for all i: When the knapsack has zero capacity, the maximum profit is zero.

#### **Step 3: Filling the DP Table**

1. For each item i from 1 to n, and for each capacity w from 0 to C:
   * If the weight of the current item (weights[i-1]) is **less than or equal** to the current capacity w, then:
     + Compute the maximum of including the current item or excluding it:
       - **Include the item**: dp[i][w] = profits[i-1] + dp[i-1][w - weights[i-1]]
       - **Exclude the item**: dp[i][w] = dp[i-1][w]
     + Store the maximum of these two options:
       - dp[i][w] = max(dp[i-1][w], profits[i-1] + dp[i-1][w - weights[i-1]])
   * If the weight of the current item exceeds the current capacity w, then the item cannot be included:
     + dp[i][w] = dp[i-1][w]

#### **Step 4: Final Answer**

1. After filling the table, the value at dp[n][C] will hold the **maximum profit** that can be achieved with n items and knapsack capacity C.

#### **Step 5: Output the Result**

1. Output the value of dp[n][C], which represents the maximum profit for the given knapsack capacity.

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

#include <iostream>

using namespace std;

int knapsack(int profits[], int weights[], int n, int capacity) {

    int dp[n+1][capacity+1];

    // Initialize dp table

    for (int i = 0; i <= n; i++) {

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

            if (i == 0 || w == 0) {

                dp[i][w] = 0;  // Base case: No items or capacity 0

            }

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

                dp[i][w] = max(dp[i-1][w], profits[i-1] + dp[i-1][w-weights[i-1]]);

            }

            else {

                dp[i][w] = dp[i-1][w];  // Item can't be included

            }

        }

    }

    return dp[n][capacity];  // Maximum profit

}

int main() {

    int n, capacity;

    // Take user input for the number of items

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

    cin >> n;

    int profits[n], weights[n];

    // Take user input for profits and weights

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

    for (int i = 0; i < n; i++) {

        cout << "Profit of item " << i+1 << ": ";

        cin >> profits[i];

    }

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

    for (int i = 0; i < n; i++) {

        cout << "Weight of item " << i+1 << ": ";

        cin >> weights[i];

    }

    // Take user input for the knapsack capacity

    cout << "Enter the knapsack capacity: ";

    cin >> capacity;

    // Call the knapsack function

    int maxProfit = knapsack(profits, weights, n, capacity);

    // Output the result

    cout << "Maximum Profit: " << maxProfit << endl;

    return 0;

}

**Sample Input and Output:**

**Input :**

Enter the number of items: 4

Enter the profits of the items:

Profit of item 1: 15

Profit of item 2: 25

Profit of item 3: 13

Profit of item 4: 23

Enter the weights of the items:

Weight of item 1: 2

Weight of item 2: 6

Weight of item 3: 12

Weight of item 4: 9

Enter the knapsack capacity: 20

**Output:**

Maximum Profit: 38

**Problem No : 10**

**Problem Title : Solving the Tower of Hanoi problem for the N disk.**

**Theory :**

The **Tower of Hanoi** is a classical problem in recursion. The puzzle consists of three rods and a number of disks of different sizes that can slide onto any rod. The puzzle starts with all disks stacked in increasing size on one rod, and the goal is to move all the disks to another rod, following these rules:

* Only one disk can be moved at a time.
* Each move consists of taking the top disk from one of the stacks and placing it on top of another stack, ensuring that no disk is placed on top of a smaller disk.
* You must use the third rod as an auxiliary to help move the disks.

The problem is recursive in nature, as solving for n disks can be reduced to solving for n-1 disks.

**Algorithm:**

#### **Recursive Algorithm to Solve the Tower of Hanoi Problem:**

1. **Base Case**: If there is only one disk (n = 1), move the disk directly from the source rod to the destination rod.
2. **Recursive Case**:
   * Move the top n-1 disks from the source rod to the auxiliary rod using the destination rod as an intermediate.
   * Move the n-th disk (largest disk) from the source rod to the destination rod.
   * Move the n-1 disks from the auxiliary rod to the destination rod using the source rod as an intermediate.

#### **Steps:**

1. Call the function to move n-1 disks from source to auxiliary using destination as helper.
2. Move the n-th disk from source to destination.
3. Call the function to move n-1 disks from auxiliary to destination using source as helper.

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

#include <iostream>

using namespace std;

// Function to solve the Tower of Hanoi problem

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

    // Base case: only one disk to move

    if (n == 1) {

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

        return;

    }

    // Move top n-1 disks from source to auxiliary

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

    // Move nth disk from source to destination

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

    // Move n-1 disks from auxiliary to destination

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

}

int main() {

    int n;

    // Input: number of disks

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

    cin >> n;

    // Call the Tower of Hanoi function

    cout << "The moves involved are:\n";

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

    return 0;

}

**Sample Input and Output :**

**Input :**

Enter the number of disks: 3

**Output :**

The moves involved 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