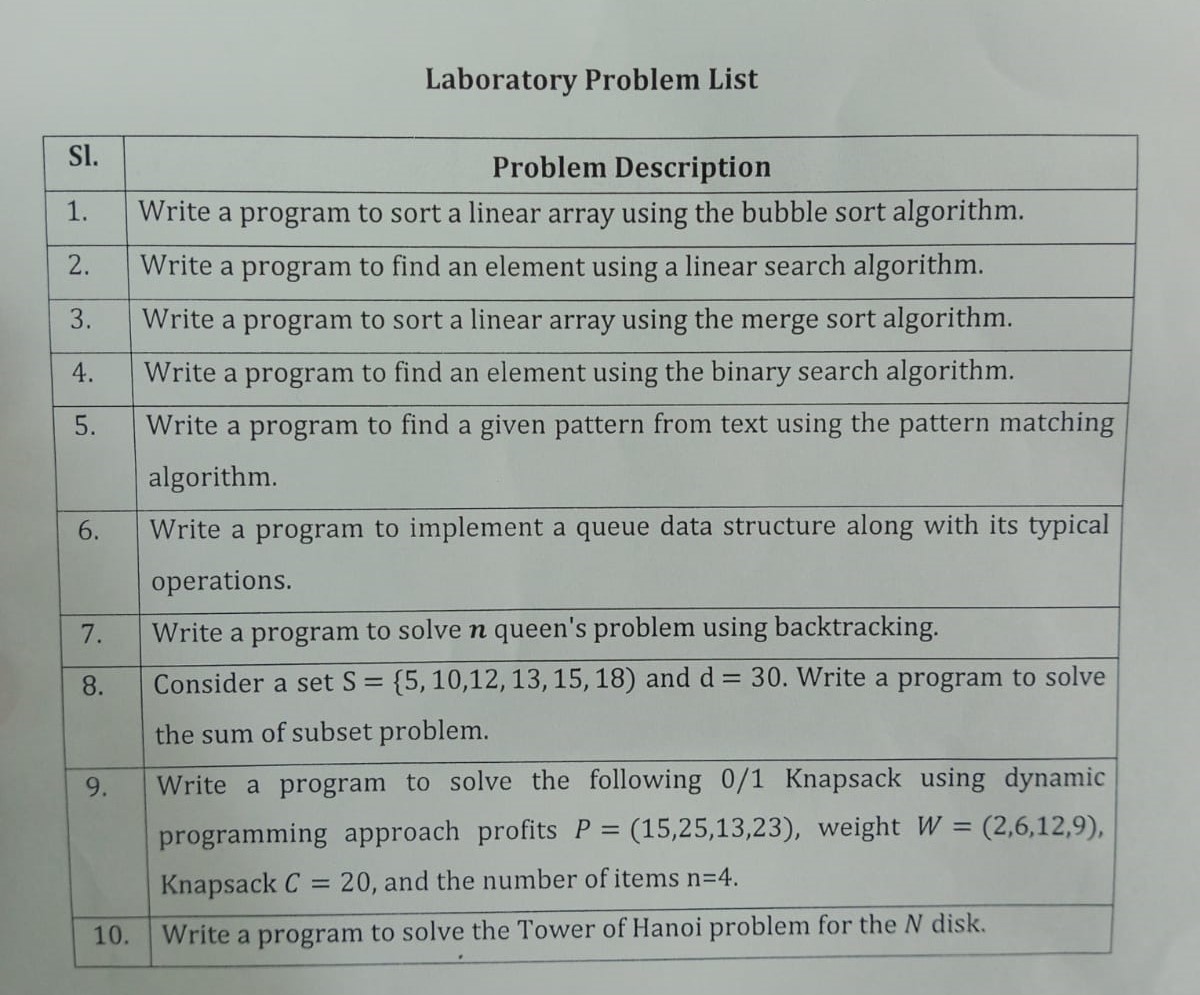
### 

### 



### **Problem Number: 01**

### **Title:** Write a program to sort a linear array using bubble sort algorithm

### **Theory :**

### Definition

Bubble Sort is a simple sorting algorithm that works by repeatedly swapping adjacent elements if they are in the wrong order. It continues this process until the entire array is sorted. The name "bubble sort" comes from the way smaller elements "bubble" to the top of the array with each pass.

### **Working Principle**

1. Comparison: Each element is compared with its adjacent element.
2. Swapping: If the left element is greater than the right, they are swapped.
3. Passes: After each complete pass, the largest element moves to its correct position.
4. Optimization: If no swaps occur in a pass, the algorithm stops early because the array is already sorted

**Algorithm:**

#### Step 1: Start

#### Step 2: Input the array

* Read the list of numbers to be sorted.

#### Step 3: Bubble Sort Process

* Outer Loop: Iterate through the array n-1 times (where n is the size of the array).
  + Initialize a swapped flag as false at the start of each pass.
  + Print the current pass number.
* Inner Loop: Compare adjacent elements and swap them if necessary.
  + Compare arr[j] and arr[j + 1].
  + If arr[j] > arr[j + 1], swap them and set swapped = true.
  + Print the status of the array after each swap.
  + If no swaps occurred in a pass, break out of the loop (optimization).
  + Print the array state at the end of each pass.

#### Step 4: Stop when the array is sorted

* If no swaps occurred in a full pass, the array is already sorted, and the algorithm terminates early.

#### Step 5: Print the sorted array

#### Step 6: End

**Source code:**

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**void bubbleSort(vector<int>& arr) {**

**int n = arr.size();**

**bool swapped;**

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

**swapped = false;**

**cout << "Pass " << i + 1 << ":" << endl;**

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

**cout << " Comparing " << arr[j] << " and " << arr[j + 1];**

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

**swap(arr[j], arr[j + 1]);**

**cout << " -> Swapped to: [";**

**for (int k = 0; k < arr.size(); k++) {**

**cout << arr[k] << (k < arr.size() - 1 ? ", " : "");**

**}**

**cout << "]" << endl;**

**swapped = true;**

**} else {**

**cout << " -> No swap" << endl;**

**}**

**}**

**cout << " Array after pass " << i + 1 << ": [";**

**for (int k = 0; k < arr.size(); k++) {**

**cout << arr[k] << (k < arr.size() - 1 ? ", " : "");**

**}**

**cout << "]" << endl;**

**if (!swapped) {**

**break;**

**}**

**}**

**}**

**int main() {**

**vector<int> arr = {5, 3, 8, 6, 2};**

**cout << "Array before sorting: [";**

**for (int i = 0; i < arr.size(); i++) {**

**cout << arr[i] << (i < arr.size() - 1 ? ", " : "");**

**}**

**cout << "]" << endl;**

**bubbleSort(arr);**

**cout << "Array after sorting: [";**

**for (int i = 0; i < arr.size(); i++) {**

**cout << arr[i] << (i < arr.size() - 1 ? ", " : "");**

**}**

**cout << "]" << endl;**

**return 0;**

**}**

**Input and output:**

**Array before sorting: [5, 3, 8, 6, 2]**

**Pass 1:**

**Comparing 5 and 3 -> Swapped to: [3, 5, 8, 6, 2]**

**Comparing 5 and 8 -> No swap**

**Comparing 8 and 6 -> Swapped to: [3, 5, 6, 8, 2]**

**Comparing 8 and 2 -> Swapped to: [3, 5, 6, 2, 8]**

**Array after pass 1: [3, 5, 6, 2, 8]**

**Pass 2:**

**Comparing 3 and 5 -> No swap**

**Comparing 5 and 6 -> No swap**

**Comparing 6 and 2 -> Swapped to: [3, 5, 2, 6, 8]**

**Array after pass 2: [3, 5, 2, 6, 8]**

**Pass 3:**

**Comparing 3 and 5 -> No swap**

**Comparing 5 and 2 -> Swapped to: [3, 2, 5, 6, 8]**

**Array after pass 3: [3, 2, 5, 6, 8]**

**Pass 4:**

**Comparing 3 and 2 -> Swapped to: [2, 3, 5, 6, 8]**

**Array after pass 4: [2, 3, 5, 6, 8]**

**Array after sorting: [2, 3, 5, 6, 8]**

### 

### 

### 

### 

### **Problem Number: 02**

### **Title:**Write a program to sort a linear array using Linear Search Algorithm

### **Theory:**Linear Search is traditionally used to find an element in an array, but it is not a sorting algorithm. However, we can use a linear approach to sort an array by repeatedly finding the smallest element and placing it at the correct position.The sorting method

### that resembles a linear search approach is Selection Sort, where we search for the smallest element in the unsorted part and swap it with the first unsorted element.

To sort an array using a linear approach:

1. Find the smallest element in the unsorted part of the array (using a linear search).
2. Swap it with the first unsorted element.
3. Move to the next index and repeat the process until the array is sorted.

This method is known as Selection Sort because it selects the smallest element from the remaining elements in each step.

## **Algorithm**

1. Start
2. Input the array.
3. Loop through the array (i from 0 to n-1):
   * Assume i is the index of the smallest element.
   * Search for the smallest element in the range [i, n-1].
   * Swap it with the element at index i.
4. Repeat until the entire array is sorted.
5. Print the sorted array.
6. End

**Source code:**

### **#include <iostream>**

### **#include <vector>**

### 

### **using namespace std;**

### **int linearSearch(const vector<int>& arr, int key) {**

### **cout << "Starting Linear Search for element " << key << ":" << endl;**

### 

### **for (int i = 0; i < arr.size(); i++) {**

### **cout << "Checking index " << i << " with value " << arr[i];**

### 

### **if (arr[i] == key) {**

### **cout << " -> Found at index " << i << endl;**

### **return i;**

### **} else {**

### **cout << " -> Not a match" << endl;**

### **}**

### **}**

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

### **return -1;**

### **}**

### 

### **int main() {**

### **vector<int> arr = {5, 3, 8, 6, 2};**

### **int key;**

### 

### **cout << "Array: [";**

### **for (int i = 0; i < arr.size(); i++) {**

### **cout << arr[i] << (i < arr.size() - 1 ? ", " : "");**

### **}**

### **cout << "]" << endl;**

### 

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

### **cin >> key;**

### **int result = linearSearch(arr, key);**

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

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

### **} else {**

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

### **}**

### **return 0;**

### **}**

### **Output:**

Array: [5, 3, 8, 6, 2]

Enter the element to search: 2

Starting Linear Search for element 2:

Checking index 0 with value 5 -> Not a match

Checking index 1 with value 3 -> Not a match

Checking index 2 with value 8 -> Not a match

Checking index 3 with value 6 -> Not a match

Checking index 4 with value 2 -> Found at index 4

Element 2 found at index 4.

### **Problem Number: 03**

### **Title:**Write a program to sort a linear array using marge sort Algorithm.

**Theory:**Merge Sort is a divide-and-conquer sorting algorithm that recursively divides an array into two halves, sorts them separately, and then merges the sorted halves back together. This method ensures that the final merged array is sorted.

It is particularly useful for sorting large datasets and maintains stable sorting, meaning it preserves the relative order of equal elements.

**Concept of Merge Sort**

1. Divide: The array is continuously split into two halves until each sub-array contains a single element (which is inherently sorted).
2. Conquer: Each pair of sorted sub-arrays is merged into a larger sorted array.
3. Combine: The process continues recursively until the entire array is merged and sorted.

**Algorithm:**

**I**nput: An unsorted array arr[] of size n.  
Output: A sorted array in ascending order.

1. If left is greater than or equal to right, return (Base Case).
2. Find the middle index: mid=(left+right)/2mid = (left + right) / 2mid=(left+right)/2
3. Recursively apply Merge Sort to the left half (arr[left...mid]).
4. Recursively apply Merge Sort to the right half (arr[mid+1...right]).
5. Merge the two sorted halves using the merge function:
   * Compare elements from both halves.
   * Insert the smaller element into the merged array.
   * Copy remaining elements (if any) from both halves.

**Source Code:**

#include <iostream>

#include <vector>

using namespace std;

void merge(vector<int>& arr, int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

vector<int> leftArr(n1);

vector<int> 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(vector<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);

cout << "After merging: [";

for (int i = 0; i < arr.size(); i++) {

cout << arr[i] << (i < arr.size() - 1 ? ", " : "");

}

cout << "]" << endl;

}

}

int main() {

vector<int> arr = {5, 3, 8, 6, 2, 7, 4, 1};

cout << "Array before sorting: [";

for (int i = 0; i < arr.size(); i++) {

cout << arr[i] << (i < arr.size() - 1 ? ", " : "");

}

cout << "]" << endl;

mergeSort(arr, 0, arr.size() - 1);

cout << "Array after sorting: [";

for (int i = 0; i < arr.size(); i++) {

cout << arr[i] << (i < arr.size() - 1 ? ", " : "");

}

cout << "]" << endl;

return 0;

}

**Input:**

Array: [5, 3, 8, 6, 2, 7, 4, 1]

**Output:**

Array after sorting: [1, 2, 3, 4, 5, 6, 7, 8]

### **Problem Number: 04**

### **Title:** Write a program to find a linear array using binary search algorithm

### **Theory :**Binary Search is an efficient searching algorithm that works on sorted arrays. It follows the divide-and-conquer approach to reduce the search space by half in each step, making it significantly faster than Linear Search.

### Working Principle:

### Find the Middle Element:

### Compute mid = (left + right) / 2.

### Compare with Target:

### If arr[mid] == target, return mid (Element Found).

### If target is greater than arr[mid], search in the right half.

### If target is smaller than arr[mid], search in the left half.

### Repeat the process until the element is found or the search space becomes empty (left > right).

### **Algorithm:**

1. Initialize left and right pointers:
   * left = 0, right = n-1.
2. While left <= right, repeat:  
   a) Compute the mid index: mid=left+(right−left)2mid = left + \frac{(right - left)}{2}mid=left+2(right−left)​ b) If arr[mid] == target, return mid.  
   c) If target > arr[mid], search in the right half (left = mid + 1).  
   d) If target < arr[mid], search in the left half (right = mid - 1).
3. If no element matches, return -1.

**Source code:**

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**int binarySearch(const vector<int>& arr, int left, int right, int target) {**

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

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

**cout << "Searching between indexes " << left << " and " << right << ", Middle index: " << mid << endl;**

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

**return mid;**

**}**

**else if (arr[mid] < target) {**

**left = mid + 1;**

**}**

**else {**

**right = mid - 1;**

**}**

**}**

**return -1;**

**}**

**int main() {**

**vector<int> arr = {2, 3, 5, 6, 8, 10, 12};**

**int target;**

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

**cin >> target;**

**int result = binarySearch(arr, 0, arr.size() - 1, target);**

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

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

**else**

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

**return 0;**

**}**

**Input & Output:**

Enter the element to search: 10

Searching between indexes 0 and 6, Middle index: 3

Searching between indexes 4 and 6, Middle index: 5

Element found at index: 5

### **Problem Number: 05**

### **Title:** Write a program to find a given pattern from text using the pattern matching algorithm**.**

### **Theory** :Pattern matching is the process of searching for a specific sequence of characters (pattern) within a larger text. It is used in search engines, text editors, plagiarism checkers, spam filtering, and DNA sequence analysis.

### Types of Pattern Matching Algorithms:

### Naive String Matching – Simple but slow.

### KMP Algorithm (Knuth-Morris-Pratt) – Efficient for long texts.

### Rabin-Karp Algorithm – Uses hashing for fast searching.

### Boyer-Moore Algorithm – Skips unnecessary comparisons.

### Naive String Matching Algorithm

### This is the simplest pattern matching method:

### Start comparing the first character of the pattern with the first character of the text.

### If they match, check the next character.

### If the whole pattern matches, return the index.

### If a mismatch occurs, shift the pattern by one position and repeat.

### Continue until the pattern is found or the text ends.

#### Example:

### Text: "ababcabc" Pattern: "abc"

### **Algorithm:**

1. Start from the first character of the text.
2. Slide the pattern over the text one character at a time.
3. For each position in the text:
   * Compare the pattern's characters with the corresponding characters in the text.
   * If a mismatch occurs, move to the next position in the text.
   * If all characters of the pattern match, print the index.
4. If the pattern is found at least once, print its position.
5. If no match is found, print "Pattern not found".

**Source Code:**

**#include <iostream>**

**#include <string>**

**using namespace std;**

**#include <string>**

**void patternMatch(string text, string pattern) {**

**int textLen = text.length();**

**int patternLen = pattern.length();**

**bool found = false;**

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

**int j;**

**for (j = 0; j < patternLen; j++) {**

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

**break;**

**}**

**}**

**if (j == patternLen) {**

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

**found = true;**

**}**

**}**

**if (!found) {**

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

**}**

**}**

**int main() {**

**string text, pattern;**

**cout << "Enter the text: ";**

**getline(cin, text);**

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

**getline(cin, pattern);**

**patternMatch(text, pattern);**

**return 0;**

**}**

### **Sample Input & Output:**

**Input:**

Enter the text: hello world

Enter the pattern to search: world

**Output:**

Pattern found at index: 6

**Input:**

Enter the text: openTo is great

Enter the pattern to search: To

**Output:**

Pattern found at index: 4

### **Problem Number: 06**

### **Title:** Write a program to implement a queue data structure along with its typical operations

### **Theory**

A queue is a linear data structure that follows the FIFO (First In, First Out) principle, meaning elements are inserted from one end (rear) and removed from the other end (front). Queues are widely used in scheduling, buffering, and managing resources in operating systems, networks, and other applications.

Queues support the following primary operations:

1. Enqueue (Insertion) – Adds an element at the rear.
2. Dequeue (Removal) – Removes an element from the front.
3. Front (Peek) – Retrieves the front element without removing it.
4. isEmpty – Checks if the queue is empty.
5. isFull – Checks if the queue is full (for fixed-size queues).

There are different types of queues:

* Simple Queue – Basic FIFO queue.
* Circular Queue – The rear wraps around when it reaches the end.
* Priority Queue – Elements are dequeued based on priority.
* Deque (Double-Ended Queue) – Insertion and deletion can happen at both ends.

**Algorithm:**

* Initialize Queue
  + Set front = -1 and rear = -1.
  + Define an array items[] of a fixed size.
* Check if the Queue is Full
  + If rear == SIZE - 1, then the queue is full.
* Check if the Queue is Empty
  + If front == -1 or front > rear, then the queue is empty.
* Enqueue (Insert an Element)
  + If the queue is full, print "Queue Overflow!"
  + Else, increment rear by 1.
  + If front == -1, set front = 0.
  + Insert the new element at items[rear].
* Dequeue (Remove an Element)
  + If the queue is empty, print "Queue Underflow!"
  + Else, print items[front] and increment front by 1.
  + If front > rear, reset front and rear to -1 (queue becomes empty).
* Peek (View Front Element)
  + If the queue is empty, print "Queue is empty!"
  + Else, print items[front].
* Display Queue Elements
  + If the queue is empty, print "Queue is empty!"
  + Else, loop from front to rear and print elements.
* End Program

**Source Code:**

**#include <iostream>**

**#define SIZE 5**

**using namespace std;**

**class Queue {**

**private:**

**int items[SIZE], front, rear;**

**public:**

**Queue() {**

**front = -1;**

**rear = -1;**

**}**

**bool isFull() {**

**return rear == SIZE - 1;**

**}**

**bool isEmpty() {**

**return front == -1 || front > rear;**

**}**

**void enqueue(int value) {**

**if (isFull()) {**

**cout << "Queue Overflow! Cannot insert " << value << endl;**

**return;**

**}**

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

**items[++rear] = value;**

**cout << "Inserted " << value << " into the queue." << endl;**

**}**

**void dequeue() {**

**if (isEmpty()) {**

**cout << "Queue Underflow! No elements to remove." << endl;**

**return;**

**}**

**cout << "Removed " << items[front] << " from the queue." << endl;**

**front++;**

**}**

**void peek() {**

**if (isEmpty()) {**

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

**return;**

**}**

**cout << "Front element: " << items[front] << endl;**

**}**

**void display() {**

**if (isEmpty()) {**

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

**return;**

**}**

**cout << "Queue elements: ";**

**for (int i = front; i <= rear; i++)**

**cout << items[i] << " ";**

**cout << endl;**

**}**

**};**

**int main() {**

**Queue q;**

**q.enqueue(10);**

**q.enqueue(20);**

**q.enqueue(30);**

**q.display();**

**q.dequeue();**

**q.display();**

**q.peek();**

**return 0;**

**}**

**Input & output:**

**input:**

enqueue(10)

enqueue(20)

enqueue(30)

display()

dequeue()

display()

peek()

**Output:**

Inserted 10 into the queue.

Inserted 20 into the queue.

Inserted 30 into the queue.

Queue elements: 10 20 30

Removed 10 from the queue.

Queue elements: 20 30

Front element: 20

### **Problem Number: 07**

### **Title:** Write a program to solve n queen’s problem using backtracking

### **Theory:**The N-Queens Problem is a classic combinatorial problem that requires placing N queens on an N × N chessboard such that no two queens attack each other. This means that no two queens can be in the same row, column, or diagonal. The problem is typically solved using the backtracking algorithm, a method that systematically explores all possible placements of queens while eliminating invalid solutions early. The algorithm works by placing a queen in a row, checking if it is safe, and then recursively attempting to place queens in the next rows. If a conflict arises, the algorithm backtracks by removing the last placed queen and trying a different column. This approach ensures that all possible board configurations are explored efficiently. The time complexity of the solution is O(N!), making it computationally expensive for large values of N. However, backtracking significantly reduces unnecessary computations by eliminating invalid placements early in the process.

### **Algorithm:**

1. Initialize the Board:
   1. Create an N × N chessboard initialized with empty cells (.).
2. Define isSafe(row, col, board, N):
   1. Check if a queen can be placed at (row, col) without being attacked:
      1. Column Check: Ensure no other queen exists in the same column.
      2. Upper Left Diagonal Check: Ensure no queen is placed in the upper-left diagonal.
      3. Upper Right Diagonal Check: Ensure no queen is placed in the upper-right diagonal.
3. Define solveNQueens(row, board, N):
   1. If row == N, a valid solution is found, print/store it.
   2. For each column col in the current row:
      1. If isSafe(row, col, board, N) returns true:
         1. Place a queen (Q) at (row, col).
         2. Recursively call solveNQueens(row + 1, board, N).
         3. If placing the queen leads to no solution, backtrack by removing the queen (Q → .).
4. Call solveNQueens(0, board, N) to start solving from the first row.
5. Print all solutions if any exist; otherwise, print "No solution found."

**Source Code:**

#include <iostream>

#include <vector>

using namespace std;

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

// Check column

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;

board[row][col] = 0;

}

}

return false;

}

void printBoard(vector<vector<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;

}

}

int main() {

int N;

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

cin >> N;

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

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

cout << "Solution for " << N << "-Queens problem:\n";

printBoard(board, N);

} else {

cout << "No solution exists for " << N << "-Queens problem.\n";

}

return 0;

}

## **Sample Input and Output:**

### **Input:**

Enter the number of queens (N): 4

**Output:**

**Solution for 4-Queens problem:**

. Q . .

. . . Q

Q . . .

. . Q .

### **Problem Number: 08**

### **Title:** Write a program to solve n queen’s problem using backtracking

### **Theory:**

### The Sum of Subset Problem is a classic combinatorial problem where we find subsets from a given set S such that the sum of their elements equals a given target sum (d).

### We use backtracking to systematically explore subsets by either:

### Including an element in the subset.

### Excluding an element from the subset.

### Backtracking when a partial sum exceeds the target d.

## **Algorithm:**

### Sort the given set in ascending order.

### Start from the first element and explore subsets recursively:

### Include the current element and check if the sum reaches d.

### If the sum exceeds d, backtrack (stop exploring that path).

### If the sum is still less than d, recursively explore the next element.

### If a valid subset is found, print it.

### Continue the process until all possible subsets are explored.

### **Source Code:**

#include <iostream>

#include <vector>

using namespace std;

void printSubset(vector<int>& subset) {

cout << "{ ";

for (int num : subset) {

cout << num << " ";

}

cout << "}" << endl;

}

void findSubsets(vector<int>& set, vector<int>& subset, int index, int currentSum, int targetSum) {

if (currentSum == targetSum) {

printSubset(subset);

return;

}

for (int i = index; i < set.size(); i++) {

if (currentSum + set[i] <= targetSum) {

subset.push\_back(set[i]);

findSubsets(set, subset, i + 1, currentSum + set[i], targetSum);

subset.pop\_back();

}

}

}

int main() {

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

int targetSum = 30;

vector<int> subset;

cout << "Subsets with sum " << targetSum << " are:" << endl;

findSubsets(set, subset, 0, 0, targetSum);

return 0;

}

## **Input & Output:**

### **Input:**

Given set = {5, 10, 12, 13, 15, 18}

Target sum = 30

**Output:**

Subsets with sum 30 are:

{ 5 10 15 }

{ 5 12 13 }

{ 12 18 }

### **Problem Number: 09**

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

## **Theory**

The 0/1 Knapsack Problem is a combinatorial optimization problem where we have a set of items, each with a weight and a profit. The goal is to determine the maximum total profit we can obtain by selecting items, subject to a weight constraint (knapsack capacity). Each item can either be included fully (1) or not at all (0)—hence the name 0/1 Knapsack.

### **Problem Statement**

Given:

* A set of n items, each having:
  + Profit (P) = {p1, p2, p3, ..., pn}
  + Weight (W) = {w1, w2, w3, ..., wn}
* A knapsack with maximum capacity C.

### **Dynamic Programming Approach**

To solve the 0/1 Knapsack problem efficiently, we use Dynamic Programming (DP) to build a table that stores solutions to subproblems. The DP approach has a time complexity of O(n\*C), making it much more efficient than the brute force recursive solution.

## **Algorithm**

### Step 1: Define the DP Table

Let dp[i][w] be the maximum profit using the first i items and a knapsack of capacity w.

### Step 2: Initialize the Table

* If there are no items (i = 0) → dp[0][w] = 0 for all w.
* If the knapsack has zero capacity (w = 0) → dp[i][0] = 0 for all i.

### Step 3: Fill the Table Using Recurrence Relation

For each item i and weight w:

* If the current item's weight is less than or equal to the current knapsack capacity:
  + We have two choices:
    1. Include the item → Profit = profit[i-1] + dp[i-1][w - weight[i-1]]
    2. Exclude the item → Profit = dp[i-1][w]
  + We take the maximum of these two: dp[i][w]=max⁡(profit[i−1]+dp[i−1][w−weight[i−1]],dp[i−1][w])dp[i][w] = \max(profit[i-1] + dp[i-1][w - weight[i-1]], dp[i-1][w])dp[i][w]=max(profit[i−1]+dp[i−1][w−weight[i−1]],dp[i−1][w])
* If the item's weight exceeds the current knapsack capacity:
  + We cannot include it, so: dp[i][w]=dp[i−1][w]dp[i][w] = dp[i-1][w]dp[i][w]=dp[i−1][w]

### Step 4: Return the Final Result

**Source Code:**

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**void printSubset(vector<int>& subset) {**

**cout << "{ ";**

**for (int num : subset) {**

**cout << num << " ";**

**}**

**cout << "}" << endl;**

**}**

**void findSubsets(vector<int>& set, vector<int>& subset, int index, int currentSum, int targetSum) {**

**if (currentSum == targetSum) {**

**printSubset(subset);**

**return;**

**}**

**for (int i = index; i < set.size(); i++) {**

**if (currentSum + set[i] <= targetSum) {**

**subset.push\_back(set[i]);**

**findSubsets(set, subset, i + 1, currentSum + set[i], targetSum);**

**subset.pop\_back();**

**}**

**}**

**}**

**int main() {**

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

**int targetSum = 30;**

**vector<int> subset;**

**cout << "Subsets with sum " << targetSum << " are:" << endl;**

**findSubsets(set, subset, 0, 0, targetSum);**

**return 0;**

**}**

**Output:**

**Maximum Profit: 63**

### **Problem Number: 10**

### **Title:** Write a program to solve the Tower of Hanoi problem for the N disk

## **Theory:**The Tower of Hanoi is a classic recursive problem that involves moving a set of disks from one rod to another while following specific rules. It consists of three rods—Source (A), Auxiliary (B), and Target (C)—and N disks of different sizes stacked in decreasing order on the source rod. The objective is to transfer all disks from Rod A to Rod C using Rod B as an auxiliary, while adhering to the following constraints: only one disk can be moved at a time, a larger disk cannot be placed on a smaller one, and disks can only be moved from the top of a stack. The problem is solved using a recursive approach, where the N-1 smaller disks are first moved to the auxiliary rod, the largest disk is moved directly to the target rod, and finally, the N-1 disks are transferred from the auxiliary rod to the target. This process continues recursively until all disks are successfully moved. The time complexity of the Tower of Hanoi problem is O(2^N), making it an exponential problem that grows rapidly with the number of disks.

## **Algorithm:**

1. Base Case:
   * If n == 1, move the disk from the source rod to the target rod and return.
2. Recursive Case:
   * Move the n-1 disks from the source rod to the auxiliary rod, using the target rod as an intermediate.
   * Move the nth (largest) disk from the source rod to the target rod.
   * Move the n-1 disks from the auxiliary rod to the target rod, using the source rod as an intermediate.

**Source code:**

**#include <iostream>**

**using namespace std;**

**void towerOfHanoi(int n, char from\_rod, char to\_rod, char aux\_rod) {**

**if (n == 1) {**

**cout << "Move disk 1 from " << from\_rod << " to " << to\_rod << endl;**

**return;**

**}**

**towerOfHanoi(n - 1, from\_rod, aux\_rod, to\_rod);**

**cout << "Move disk " << n << " from " << from\_rod << " to " << to\_rod << endl;**

**towerOfHanoi(n - 1, aux\_rod, to\_rod, from\_rod);**

**}**

**int main() {**

**int N;**

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

**cin >> N;**

**towerOfHanoi(N, 'A', 'C', 'B');**

**return 0;**

**}**

**Input & Output:**

**Input:**

Enter number of disks: 3

**Output:**

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