

**Pabna University of Science and Technology, Pabna**

**Information and communication Engineering**

Course Name: Data Structure and Algorithms Sessional

Couser Code: ICE-2202

Lab Report

SUBMITTED BY:

Mushfiqur Rahman Toki

Roll: 220629

Session: 2021-2022

[Cite your source here.]

SUBMITTED TO:

[Md. Anwar Hossain](https://www.pust.ac.bd/academic/departments/dept_teachers/dept_teachers_profile/100072)

Professor

Department of Information and Communication Engineering, PUST

Submission Data: 24.02.2025

|  |  |
| --- | --- |
| **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. | |

**INDEX**

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

**Theory:**

Bubble Sort is a comparison-based sorting algorithm that repeatedly steps through the list of elements to be sorted, compares adjacent elements, and swaps them if they are in the wrong order. The process repeats until the array is sorted. It gets its name because the largest unsorted element "bubbles" to its correct position with each iteration.

This algorithm is simple but not the most efficient for large datasets. Its worst-case and average-case time complexity is O(n2)O(n^2)O(n2), which makes it inefficient compared to other algorithms like Merge Sort or Quick Sort.

**Key Steps:**

1. Traverse the entire array from the start to the end.
2. Compare adjacent elements and swap them if they are in the wrong order.
3. After each full pass, the largest element is moved to the end.
4. Repeat the process for the remaining unsorted part of the array.
5. Stop when no swaps are needed, indicating that the array is sorted.

**Bubble Sort Characteristics:**

* **Time Complexity**:
  + Worst Case: O(n2)O(n^2)O(n2)
  + Best Case: O(n)O(n)O(n) (when the array is already sorted)
  + Average Case: O(n2)O(n^2)O(n2)
* **Space Complexity**: O(1)O(1)O(1) (since it’s an in-place sorting algorithm)
* **Stable Sorting**: Bubble Sort is stable, meaning that it maintains the relative order of elements with equal values.

**Algorithm:**

1. Loop through the array n times, where n is the size of the array.
2. In each iteration, compare adjacent elements. If the current element is greater than the next element, swap them.
3. Continue looping until no swaps are needed in an iteration, meaning the array is sorted.

**Source Code in C++:**#include <iostream>

using namespace std;

// Function to perform bubble sort

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

    // Traverse through all array elements

    for (int i = 0; i < n-1; i++) {

        bool swapped = false; // Flag to check if any swapping happens

        // Last i elements are already sorted

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

            // Swap if the element is greater than the next element

            if (arr[j] > arr[j+1]) {

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

                swapped = true;

            }

        }

        // If no elements were swapped, break the loop early

        if (!swapped)

            break;

    }

}

// Function to print the array

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

    for (int i = 0; i < n; i++) {

        cout << arr[i] << " ";

    }

    cout << endl;

}

int main() {

    // Input array

    int arr[] = {64, 34, 25, 12, 22, 11, 90};

    int n = sizeof(arr)/sizeof(arr[0]);

    cout << "Unsorted array: ";

    printArray(arr, n);

    // Call bubbleSort function

    bubbleSort(arr, n);

    cout << "Sorted array: ";

    printArray(arr, n);

    return 0;

}

### Input:

The input array is provided in the code:

int arr[] = {64, 34, 25, 12, 22, 11, 90};

You can modify the array in the code or prompt the user to enter input dynamically.

### Output:

The program will print the unsorted array first, followed by the sorted array:

Unsorted array: 64 34 25 12 22 11 90

Sorted array: 11 12 22 25 34 64 90

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

**Theory:**

Linear Search is one of the simplest searching algorithms. It checks every element of the array sequentially from the first to the last until it finds the target element or reaches the end of the array. The algorithm works well for small datasets or unsorted arrays, but its time complexity makes it inefficient for large datasets.

In linear search:

* The algorithm starts at the first element of the array.
* It compares the target element with the current element in the array.
* If a match is found, the index of the element is returned.
* If the end of the array is reached without finding the target, the algorithm returns an indication that the element is not found.

**Key Characteristics:**

* **Time Complexity**:
  + Worst Case: O(n)O(n)O(n) (when the element is at the end or not present).
  + Best Case: O(1)O(1)O(1) (when the element is the first element).
  + Average Case: O(n)O(n)O(n).
* **Space Complexity**: O(1)O(1)O(1) (since it’s an in-place searching algorithm).
* **Works for unsorted arrays**: Unlike more advanced algorithms like binary search, linear search works on both sorted and unsorted arrays.

**Algorithm:**

1. Start from the first element of the array.
2. Compare the current element with the target element.
3. If they match, return the index of the element.
4. If the element is not found, move to the next element.
5. Repeat the process until either the element is found or the end of the array is reached Top of For

**Source Code**

#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 the index where the element is found

        }

    }

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

}

int main() {

    int n, key;

    // Taking input for array size

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

    cin >> n;

    int arr[n];

    // Taking input for array elements

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

    for (int i = 0; i < n; i++) {

        cin >> arr[i];

    }

    // Taking input for the key to search

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

    cin >> key;

    // Performing linear search

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

    // Displaying result

    if (result != -1)

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

    else

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

    return 0;

}

### ****Sample Input 1****

Enter the number of elements: 5

Enter 5 elements: 10 20 30 40 50

Enter the element to search: 30

### ****Sample Output 1****

Element found at index 2

## **Title:** Sorting a Linear Array Using Merge Sort in C++

### ****Theory****

Merge Sort is a **divide and conquer** sorting algorithm that:

1. **Divides** the array into two halves recursively until each subarray has only one element.
2. **Merges** the sorted subarrays back together to form a sorted array.

#### **Key Features:**

* **Time Complexity:**
  + **Best case:** O(n log n)
  + **Worst case:** O(n log n)
  + **Average case:** O(n log n)
* **Space Complexity:** O(n) (additional space needed for merging)
* **Stable Sort:** Maintains the order of duplicate element
* **Efficient for large datasets.**

## **Algorithm**

1. If the array has **one or zero** elements, return (already sorted).
2. Divide the array into **two halves** (left and right).
3. Recursively call Merge Sort on both halves.
4. Merge the two sorted halves:
   * Compare elements from both halves.
   * Insert the smaller element into the sorted array.
   * Repeat until all elements are merged.
5. The sorted array is now ready.

## **Source Code**

#include <iostream>

using namespace std;

// Function to merge two halves of an array

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

    int n1 = mid - left + 1; // Size of left subarray

    int n2 = right - mid;    // Size of right subarray

    int leftArr[n1], rightArr[n2];

    // Copy data to temp arrays

    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; // Initial indexes

    // Merge the temp arrays back into the main array

    while (i < n1 && j < n2) {

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

            arr[k] = leftArr[i];

            i++;

        } else {

            arr[k] = rightArr[j];

            j++;

        }

        k++;

    }

    // Copy remaining elements of leftArr[], if any

    while (i < n1) {

        arr[k] = leftArr[i];

        i++;

        k++;

    }

    // Copy remaining elements of rightArr[], if any

    while (j < n2) {

        arr[k] = rightArr[j];

        j++;

        k++;

    }

}

// Function to perform Merge Sort

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

    if (left < right) {

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

        // Recursively sort both halves

        mergeSort(arr, left, mid);

        mergeSort(arr, mid + 1, right);

        // Merge sorted halves

        merge(arr, left, mid, right);

    }

}

int main() {

    int n;

    // Taking input for array size

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

    cin >> n;

    int arr[n];

    // Taking input for array elements

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

    for (int i = 0; i < n; i++) {

        cin >> arr[i];

    }

    // Sorting the array using Merge Sort

    mergeSort(arr, 0, n - 1);

    // Displaying sorted array

    cout << "Sorted array: ";

    for (int i = 0; i < n; i++) {

        cout << arr[i] << " ";

    }

    cout << endl;

    return 0;

}

## **Input & Output**

### ****Sample Input****

Enter the number of elements: 5

Enter 5 elements: 50 20 40 10 30

### ****Sample Output****

Sorted array: 10 20 30 40 50

## **Title**: Binary Search Algorithm in C++

## **Theory**:

Binary Search is an efficient searching algorithm used to find an element in a sorted array. It works by repeatedly dividing the array into two halves and comparing the target value with the middle element. If the target is found, the search ends. Otherwise, the search continues in the left or right half, depending on whether the target is smaller or greater than the middle element.

### ****Time Complexity****:

* **Best Case**: O(1)O(1)O(1) (Element found at the middle)
* **Average Case**: O(log⁡n)O(\log n)O(logn) (Dividing array in half each time)
* **Worst Case**: O(log⁡n)O(\log n)O(logn) (Element not present)

### ****Space Complexity****:

* **Iterative**: O(1)O(1)O(1)
* **Recursive**: O(log⁡n)O(\log n)O(logn) (Due to recursion stack)

## **Algorithm**:

1. Start with two pointers, low (beginning of the array) and high (end of the array).
2. Compute the middle index: mid=low+(high−low)/2mid = low + (high - low) / 2mid=low+(high−low)/2
3. Compare arr[mid] with the target value:
   * If arr[mid] equals the target, return mid (element found).
   * If arr[mid] is greater than the target, search in the left half (high = mid - 1).
   * If arr[mid] is smaller than the target, search in the right half (low = mid + 1).
4. Repeat until low > high, which means the element is not present.

## **C++ Source Code (Iterative Approach)**:

#include <iostream>

using namespace std;

// Function for Binary Search (Iterative)

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

    int low = 0, high = size - 1;

    while (low <= high) {

        int mid = low + (high - low) / 2; // Prevents overflow

        if (arr[mid] == target)

            return mid; // Element found

        else if (arr[mid] < target)

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

        else

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

    }

    return -1; // Element not found

}

int main() {

    int arr[] = {2, 5, 8, 12, 16, 23, 38, 45, 56, 72}; // Sorted array

    int size = sizeof(arr) / sizeof(arr[0]);

    int target;

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

    cin >> target;

    int result = binarySearch(arr, size, target);

    if (result != -1)

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

    else

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

    return 0;

}

## **Input/Output Examples**

### ****Example 1 (Element Found)****

#### **Input**:

Enter the element to search: 23

#### **Output**:

Element found at index: 5

### ****Example 2 (Element Not Found)****

#### **Input**:

Enter the element to search: 50

#### **Output**:

Element not found

# **Pattern Matching Algorithm in C++**

## **Title**: Pattern Matching Using Naïve Algorithm in C++

## **Theory**:

Pattern matching is a technique used to search for a specific substring (pattern) within a larger text. Various algorithms are used for pattern matching, such as:

1. **Naïve Algorithm** (Brute-force method)
2. **Knuth-Morris-Pratt (KMP) Algorithm**
3. **Rabin-Karp Algorithm**
4. **Boyer-Moore Algorithm**

In this example, we will implement the **Naïve Algorithm**, which is simple but effective for small text sizes.

### ****Time Complexity****:

* **Best Case**: O(n)O(n)O(n) (When mismatches occur early)
* **Worst Case**: O(m×n)O(m \times n)O(m×n) (If the pattern appears frequently)

Where:

* nnn = Length of the text
* mmm = Length of the pattern

### ****Space Complexity****:

* O(1)O(1)O(1) (No additional memory usage)

## **Algorithm (Naïve Approach)**:

1. Take input for the **text** and **pattern**.
2. Iterate over the **text** from index i = 0 to i = n - m (where nnn is text length and mmm is pattern length).
3. For each position, compare the substring with the pattern:
   * If all characters match, store the index.
   * If a mismatch occurs, move to the next position.
4. Print all found positions.
5. If no match is found, print "Pattern not found".
6. #include <iostream>
7. #include <string>
8. using namespace std;
9. // Function to find the pattern in the given text
10. void patternMatching(string text, string pattern) {
11. int n = text.length();
12. int m = pattern.length();
13. bool found = false;
14. for (int i = 0; i <= n - m; i++) {
15. int j;
16. // Check if the pattern matches the substring starting at i
17. for (j = 0; j < m; j++) {
18. if (text[i + j] != pattern[j])
19. break;
20. }
21. if (j == m) { // If full pattern matched
22. cout << "Pattern found at index: " << i << endl;
23. found = true;
24. }
25. }
26. if (!found)
27. cout << "Pattern not found" << endl;
28. }
29. int main() {
30. string text, pattern;
31. // Taking input from user
32. cout << "Enter the text: ";
33. getline(cin, text);
34. cout << "Enter the pattern to search: ";
35. getline(cin, pattern);
36. // Call pattern matching function
37. patternMatching(text, pattern);
38. return 0;
39. }

## **Input/Output Examples**

### ****Example 1 (Pattern Found)****

#### **Input**:

Enter the text: abcdeabcdfabc

Enter the pattern to search: abc

#### **Output**:

Pattern found at index: 0

Pattern found at index: 5

Pattern found at index: 10

### ****Example 2 (Pattern Not Found)****

#### **Input**:

Enter the text: hello world

Enter the pattern to search: abc

#### **Output**:

Pattern not found

# **Queue Data Structure Implementation in C++**

## **Title**: Queue Data Structure and Its Operations in C++

## **Theory**:

A **queue** is a **linear data structure** that follows the **FIFO (First In, First Out)** principle. The element inserted first is removed first, similar to a queue of people waiting in a line.

### ****Types of Queues****:

1. **Simple Queue** – Basic FIFO structure.
2. **Circular Queue** – The last position is connected to the first.
3. **Priority Queue** – Elements are dequeued based on priority.
4. **Double-Ended Queue (Deque)** – Insertion and deletion can happen from both ends.

### ****Basic Operations****:

1. **Enqueue (Insertion)** – Adds an element to the rear.
2. **Dequeue (Deletion)** – Removes an element from the front.
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.

### ****Time Complexity****:

* **Enqueue**: O(1)O(1)O(1)
* **Dequeue**: O(1)O(1)O(1)
* **Front (Peek)**: O(1)O(1)O(1)
* **isEmpty / isFull**: O(1)O(1)O(1)

### ****Space Complexity****:

* O(n)O(n)O(n), where nnn is the maximum size of the queue.

## **Algorithm for Queue Operations**:

1. **Initialize** the queue with a fixed size.
2. **Enqueue Operation**:
   * If the queue is full, print "Queue is full".
   * Otherwise, insert the element at the rear and update rear.
3. **Dequeue Operation**:
   * If the queue is empty, print "Queue is empty".
   * Otherwise, remove the element from the front and update front.
4. **Front Operation**:
   * If the queue is not empty, return the front element.
5. **Check if Empty**:
   * If front > rear, the queue is empty.
6. **Check if Full**:
   * If rear == size - 1, the queue is full.
   * Source code
7. #include <iostream>
8. using namespace std;
9. #define SIZE 5  // Maximum queue size
10. class Queue {
11. private:
12. int arr[SIZE];
13. int front, rear;
14. public:
15. // Constructor
16. Queue() {
17. front = -1;
18. rear = -1;
19. }
20. // Check if the queue is empty
21. bool isEmpty() {
22. return (front == -1 || front > rear);
23. }
24. // Check if the queue is full
25. bool isFull() {
26. return (rear == SIZE - 1);
27. }
28. // Enqueue operation
29. void enqueue(int value) {
30. if (isFull()) {
31. cout << "Queue is full! Cannot insert " << value << endl;
32. return;
33. }
34. if (front == -1) front = 0; // Initialize front if adding first element
35. arr[++rear] = value;
36. cout << "Inserted: " << value << endl;
37. }
38. // Dequeue operation
39. void dequeue() {
40. if (isEmpty()) {
41. cout << "Queue is empty! Nothing to remove." << endl;
42. return;
43. }
44. cout << "Removed: " << arr[front] << endl;
45. front++;
46. }
47. // Peek operation (Front element)
48. void peek() {
49. if (isEmpty()) {
50. cout << "Queue is empty! No front element." << endl;
51. return;
52. }
53. cout << "Front element: " << arr[front] << endl;
54. }
55. // Display queue elements
56. void display() {
57. if (isEmpty()) {
58. cout << "Queue is empty!" << endl;
59. return;
60. }
61. cout << "Queue elements: ";
62. for (int i = front; i <= rear; i++) {
63. cout << arr[i] << " ";
64. }
65. cout << endl;
66. }
67. };
68. int main() {
69. Queue q;
70. int choice, value;
71. while (true) {
72. cout << "\nQueue Operations:\n";
73. cout << "1. Enqueue\n2. Dequeue\n3. Peek\n4. Display\n5. Exit\n";
74. cout << "Enter your choice: ";
75. cin >> choice;
76. switch (choice) {
77. case 1:
78. cout << "Enter value to insert: ";
79. cin >> value;
80. q.enqueue(value);
81. break;
82. case 2:
83. q.dequeue();
84. break;
85. case 3:
86. q.peek();
87. break;
88. case 4:
89. q.display();
90. break;
91. case 5:
92. cout << "Exiting program..." << endl;
93. return 0;
94. default:
95. cout << "Invalid choice! Try again." << endl;
96. }
97. }
98. return 0;
99. }
100. #include <iostream>
101. using namespace std;
102. #define SIZE 5  // Maximum queue size
103. class Queue {
104. private:
105. int arr[SIZE];
106. int front, rear;
107. public:
108. // Constructor
109. Queue() {
110. front = -1;
111. rear = -1;
112. }
113. // Check if the queue is empty
114. bool isEmpty() {
115. return (front == -1 || front > rear);
116. }
117. // Check if the queue is full
118. bool isFull() {
119. return (rear == SIZE - 1);
120. }
121. // Enqueue operation
122. void enqueue(int value) {
123. if (isFull()) {
124. cout << "Queue is full! Cannot insert " << value << endl;
125. return;
126. }
127. if (front == -1) front = 0; // Initialize front if adding first element
128. arr[++rear] = value;
129. cout << "Inserted: " << value << endl;
130. }
131. // Dequeue operation
132. void dequeue() {
133. if (isEmpty()) {
134. cout << "Queue is empty! Nothing to remove." << endl;
135. return;
136. }
137. cout << "Removed: " << arr[front] << endl;
138. front++;
139. }
140. // Peek operation (Front element)
141. void peek() {
142. if (isEmpty()) {
143. cout << "Queue is empty! No front element." << endl;
144. return;
145. }
146. cout << "Front element: " << arr[front] << endl;
147. }
148. // Display queue elements
149. void display() {
150. if (isEmpty()) {
151. cout << "Queue is empty!" << endl;
152. return;
153. }
154. cout << "Queue elements: ";
155. for (int i = front; i <= rear; i++) {
156. cout << arr[i] << " ";
157. }
158. cout << endl;
159. }
160. };
161. int main() {
162. Queue q;
163. int choice, value;
164. while (true) {
165. cout << "\nQueue Operations:\n";
166. cout << "1. Enqueue\n2. Dequeue\n3. Peek\n4. Display\n5. Exit\n";
167. cout << "Enter your choice: ";
168. cin >> choice;
169. switch (choice) {
170. case 1:
171. cout << "Enter value to insert: ";
172. cin >> value;
173. q.enqueue(value);
174. break;
175. case 2:
176. q.dequeue();
177. break;
178. case 3:
179. q.peek();
180. break;
181. case 4:
182. q.display();
183. break;
184. case 5:
185. cout << "Exiting program..." << endl;
186. return 0;
187. default:
188. cout << "Invalid choice! Try again." << endl;
189. }
190. }
191. return 0;
192. }

## **Input/Output Examples**

### ****Example 1: Queue Operations****

#### **Input (User Choices)**

CopyEdit

1

10

1

20

1

30

4

2

4

3

5

#### **Output**

yaml

CopyEdit

Inserted: 10

Inserted: 20

Inserted: 30

Queue elements: 10 20 30

Removed: 10

Queue elements: 20 30

Front element: 20

Exiting program...

### ****Example 2: Trying to Dequeue from an Empty Queue****

#### **Input**

CopyEdit

2

#### **Output**

# **N-Queens Problem Using Backtracking in C++**

## **Title**: Solving the N-Queens Problem Using Backtracking in C++

## **Theory**:

The **N-Queens Problem** is a classic combinatorial problem where we must place **N queens** on an **N×N chessboard** so that no two queens attack each other. This means:

1. No two queens share the same **row**.
2. No two queens share the same **column**.
3. No two queens share the same **diagonal**.

To solve this, we use **backtracking**, a technique that explores all possibilities and backtracks when an invalid configuration is found.

### ****Time Complexity****:

* **Worst Case**: O(N!)O(N!)O(N!) (Since we explore all possible board configurations)
* **Average Case**: O(N2)O(N^2)O(N2) (Due to pruning of invalid states)

### ****Space Complexity****:

* O(N2)O(N^2)O(N2) (For storing the board)

## **Algorithm (Backtracking Approach)**:

1. Start by placing the **first queen** in the first row.
2. Place the next queen in the next row in a column where it is **not attacked**.
3. If no valid position is found in a row, **backtrack** to the previous row and try a different position.
4. Repeat until all queens are placed.
5. If a valid arrangement is found, print the solution.
6. #include <iostream>
7. #include <vector>
8. using namespace std;
9. #define N 8  // Change this value for different board sizes
10. // Function to print the chessboard
11. void printSolution(vector<vector<int>>& board) {
12. for (int i = 0; i < N; i++) {
13. for (int j = 0; j < N; j++) {
14. cout << (board[i][j] ? "Q " : ". ");
15. }
16. cout << endl;
17. }
18. cout << endl;
19. }
20. // Function to check if a queen can be placed at board[row][col]
21. bool isSafe(vector<vector<int>>& board, int row, int col) {
22. // Check column for conflicts
23. for (int i = 0; i < row; i++)
24. if (board[i][col])
25. return false;
26. // Check upper-left diagonal
27. for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)
28. if (board[i][j])
29. return false;
30. // Check upper-right diagonal
31. for (int i = row, j = col; i >= 0 && j < N; i--, j++)
32. if (board[i][j])
33. return false;
34. return true; // No conflicts, placement is safe
35. }
36. // Function to solve the N-Queens problem using backtracking
37. bool solveNQueens(vector<vector<int>>& board, int row) {
38. if (row >= N) { // All queens placed successfully
39. printSolution(board);
40. return true; // Change to false to print only one solution
41. }
42. bool foundSolution = false;
43. for (int col = 0; col < N; col++) {
44. if (isSafe(board, row, col)) {
45. board[row][col] = 1; // Place queen
46. // Recursive call to place next queen
47. foundSolution = solveNQueens(board, row + 1) || foundSolution;
48. // Backtrack (Remove queen)
49. board[row][col] = 0;
50. }
51. }
52. return foundSolution;
53. }
54. int main() {
55. vector<vector<int>> board(N, vector<int>(N, 0));
56. if (!solveNQueens(board, 0))
57. cout << "No solution exists for N = " << N << endl;
58. return 0;
59. }

## **Input/Output Examples**

### ****Example 1 (For N = 4)****

#### **Output**:

css

CopyEdit

. Q . .

. . . Q

Q . . .

. . Q .

. . Q .

Q . . .

. . . Q

. Q . .

(The program prints all valid board configurations.)

### ****Example 2 (For N = 8)****

#### **Output**:

One of the possible solutions:

css

CopyEdit

Q . . . . . . .

. . . . Q . . .

. . . . . . . Q

. . . . . Q . .

. . Q . . . . .

. . . . . . Q .

. Q . . . . . .

. . . Q . . . .

# **Sum of Subset Problem Using Backtracking in C++**

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

## **Theory**:

The **Sum of Subset Problem** is a classic combinatorial problem in which we need to determine all subsets of a given set **S** whose sum equals a specified target value.

### ****Problem Statement****:

Given a set **S = {5, 10, 12, 13, 15, 18}**, find all subsets that sum up to a given **target sum**.

### ****Backtracking Approach****:

1. Start with an empty subset.
2. Add elements one by one while ensuring that the sum does not exceed the target.
3. If the sum equals the target, print the subset.
4. If adding an element exceeds the target, backtrack and try another possibility.

### ****Time Complexity****:

* In the worst case, we explore all 2N2^N2N subsets, making the complexity **O(2N)O(2^N)O(2N)**.

## **Algorithm for Sum of Subset Problem**:

1. **Sort the set** (not necessary but helps in optimization).
2. **Use recursion and backtracking**:
   * Include an element and move forward.
   * Exclude the element and move to the next.
3. If at any point the **sum matches the target**, print the subset.
4. If the sum exceeds the target, backtrack.
5. Repeat until all subsets are explored.

## **C++ Source Code**:

#include <iostream>

#include <vector>

using namespace std;

void findSubsets(vector<int>& set, vector<int>& subset, int index, int sum, int target) {

    // If sum matches the target, print the subset

    if (sum == target) {

        cout << "{ ";

        for (int num : subset)

            cout << num << " ";

        cout << "}" << endl;

        return;

    }

    // If index is out of bounds or sum exceeds target, stop recursion

    if (index >= set.size() || sum > target)

        return;

    // Include current element in subset and recurse

    subset.push\_back(set[index]);

    findSubsets(set, subset, index + 1, sum + set[index], target);

    // Exclude current element (backtrack) and recurse

    subset.pop\_back();

    findSubsets(set, subset, index + 1, sum, target);

}

int main() {

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

    vector<int> subset;

    int target;

    cout << "Enter the target sum: ";

    cin >> target;

    cout << "Subsets with sum " << target << ":" << endl;

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

    return 0;

}

## **Input/Output Examples**

### ****Example 1****

#### **Input**:

Enter the target sum: 25

#### **Output**:

Subsets with sum 25:

{ 10 15 }

{ 12 13 }

{ 5 10 12 }

### ****Example 2****

#### **Input**:

Enter the target sum: 30

#### **Output**:

Subsets with sum 30:

{ 12 18 }

{ 5 10 15 }

## **Title**: Solving the Tower of Hanoi Problem Using Recursion in C++

## **Theory**:

The **Tower of Hanoi** is a classic mathematical problem that involves moving a set of disks from one peg to another using a third peg, following these rules:

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 peg.
3. The goal is to move **all disks** from the source peg to the destination peg using an auxiliary peg.

### ****Number of Moves Required****:

For **N disks**, the minimum number of moves required to solve the Tower of Hanoi problem is given by:

M=2N−1M = 2^N - 1M=2N−1

### ****Recursive Approach****:

* Move **(N-1) disks** from **Source → Auxiliary** peg.
* Move the **Nth disk** directly to the **Destination** peg.
* Move the **(N-1) disks** from **Auxiliary → Destination** peg.

## **Algorithm (Recursive Approach)**:

1. **Base Case**: If only one disk is left, move it directly from source to destination.
2. **Recursive Step**:
   * Move **N-1** disks from source to auxiliary using destination as a helper.
   * Move the **Nth disk** from source to destination.
   * Move **N-1** disks from auxiliary to destination using source as a helper.

## **C++ Source Code**:

#include <iostream>

using namespace std;

// Function to solve Tower of Hanoi

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

    if (n == 1) {

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

        return;

    }

    // Move (N-1) disks from source to auxiliary peg using destination

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

    // Move the nth disk from source to destination

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

    // Move (N-1) disks from auxiliary to destination using source

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

}

int main() {

    int n;

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

    cin >> n;

    cout << "\nSteps to solve Tower of Hanoi for " << n << " disks:\n";

    towerOfHanoi(n, 'A', 'B', 'C'); // A = Source, B = Auxiliary, C = Destination

    cout << "\nTotal moves required: " << (1 << n) - 1 << endl; // 2^N - 1

    return 0;

}

## **Input/Output Examples**

### ****Example 1 (For N = 3)****

#### **Input**:

Enter the number of disks: 3

#### **Output**:

Steps to solve Tower of Hanoi for 3 disks:

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

Total moves required: 7

### ****Example 2 (For N = 4)****

#### **Input**:

Enter the number of disks: 4

#### **Output**

Steps to solve Tower of Hanoi for 4 disks:

Move disk 1 from A to B

Move disk 2 from A to C

Move disk 1 from B to C

Move disk 3 from A to B

Move disk 1 from C to A

Move disk 2 from C to B

Move disk 1 from A to B

Move disk 4 from A to C

Move disk 1 from B to C

Move disk 2 from B to A

Move disk 1 from C to A

Move disk 3 from B to C

Move disk 1 from A to B

Move disk 2 from A to C

Move disk 1 from B to C

Total moves required: 15