

**Index**

| 01 | Write a program to sort a linear array using the bubble sort algorithm. |
| --- | --- |
| 02 | Write a program to find an element using a linear search algorithm. |
| 03 | Write a program to find an element using a linear search algorithm. |
| 04 | Write a program to find an element using the binary search algorithm |
| 05 | Write a program to find a given pattern from text using the pattern matching algorithm. |
| 06 | Write a program to implement a queue data structure along with its typical operations. |
| 07 | Write a program to solve n queen's problem using backtracking. |
| 08 | Consider a set S = {5,10,12,13,15,18} and d = 30. Write a program to solve the sum of subset problem. |
| 09 | Write a 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. |

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

#### 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. The pass through the list is repeated until the list is sorted. It has a time complexity of in the worst and average cases.

#### Algorithm:

1. Start with the first element of the array.
2. Compare it with the next element.
3. If the current element is greater than the next element, swap them.
4. Move to the next pair of elements and repeat steps 2-3.
5. Continue this process until no swaps are needed in a full pass through the array.

#### Source Code (C++):

#include <iostream>

using namespace std;

void bubbleSort(int arr[], int n)

{

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

{

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

{

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

{

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

}

}

}

}

int main()

{

int n;

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

cin >> n;

int arr[n];

cout << "Enter the elements: ";

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

{

cin >> arr[i];

}

bubbleSort(arr, n);

cout << "Sorted array: ";

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

{

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

}

return 0;

}

#### Input:

**Enter the number of elements: 7**

**Enter the elements: 64 34 25 12 22 11 90**

#### Output:

**Sorted array: 11 12 22 25 34 64 90**

### Lab 2 : Write a program to find an element using a linear search algorithm.

#### Title: Finding an Element Using Linear Search Algorithm.

#### Theory:

Linear Search is a method for finding a particular value in a list. It checks each element of the list one by one until a match is found or the whole list has been searched. It has a time complexity of .

#### Algorithm:

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

#### 

#### Source Code (C++):

#include <iostream>

using namespace std;

int linearSearch(int arr[], int n, int target) {

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

if (arr[i] == target) {

return i;

}

}

return -1;

}

int main() {

int n;

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

cin >> n;

int arr[n];

cout << "Enter the elements: ";

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

cin >> arr[i];

}

int target;

cout << "Enter the target element: ";

cin >> target;

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

if (result == -1) {

cout << "Element not found!";

} else {

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

}

return 0;

}

#### Input:

**Enter the number of elements: 5**

**Enter the elements: 10 20 30 40 50**

**Enter the target element: 30**

#### Output:

**Element found at index: 2**

### Lab 3 : Write a program to sort a linear array using the merge sort algorithm.

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

#### Theory:

Merge Sort is a divide-and-conquer algorithm that divides the input array into two halves, recursively sorts them, and then merges the two sorted halves. It has a time complexity of .

#### Algorithm:

1.Divide the array into two halves.

2.Recursively sort each half.

3.Merge the two sorted halves into a single sorted array.

#### Source Code (C++):

#include <iostream>

using namespace std;

void merge(int arr[], int l, int m, int r) {

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

for (int i = 0; i < n1; i++) L[i] = arr[l + i];

for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

int i = 0, j = 0, k = l;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) arr[k++] = L[i++];

else arr[k++] = R[j++];

}

while (i < n1) arr[k++] = L[i++];

while (j < n2) arr[k++] = R[j++];

}

void mergeSort(int arr[], int l, int r) {

if (l < r) {

int m = l + (r - l) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

int main() {

int n;

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

cin >> n;

int arr[n];

cout << "Enter the elements: ";

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

cin >> arr[i];

}

mergeSort(arr, 0, n - 1);

cout << "Sorted array: ";

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

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

}

return 0;

}

#### Input:

#### Enter the number of elements: 7

#### Enter the elements: 38 27 43 3 9 82 10

#### 

#### Output:

#### Sorted array: 3 9 10 27 38 43 82

### 

### 

### Lab 4 : Write a program to find an element using the binary search algorithm.

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

#### Theory:

Binary Search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing the search interval in half. It has a time complexity of .

#### Algorithm:

1. Start with the middle element of the sorted array.
2. If the target value is equal to the middle element, return its index.

3. If the target value is less than the middle element, narrow the search to the lower   
 half.

4. If the target value is greater than the middle element, narrow the search to the   
 upper half.

5. Repeat steps 1-4 until the target is found or the search interval is empty.

#### Source Code (C++):

#include <iostream>

using namespace std;

int binarySearch(int arr[], int l, int r, int target) {

while (l <= r) {

int m = l + (r - l) / 2;

if (arr[m] == target) return m;

if (arr[m] < target) l = m + 1;

else r = m - 1;

}

return -1;

}

int main() {

int n;

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

cin >> n;

int arr[n];

cout << "Enter the sorted elements: ";

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

cin >> arr[i];

}

int target;

cout << "Enter the target element: ";

cin >> target;

int result = binarySearch(arr, 0, n - 1, target);

if (result == -1) {

cout << "Element not found!";

} else {

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

}

return 0;

}

#### Input:

#### Enter the number of elements: 5

#### Enter the sorted elements: 2 3 4 10 40

#### Enter the target element: 10

#### 

#### Output:

#### Element found at index: 3

### 

### 

### 

### Lab 5 : Write a program to find a given pattern from text using the pattern matching algorithm.

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

#### Theory:

Pattern Matching is used to find the occurrence of a pattern within a text. The Naive Algorithm checks for all possible positions of the pattern in the text. It has a time complexity of

, where is the text length and is the pattern length.

#### Algorithm:

1. Slide the pattern over the text one by one.
2. For each position, check if the pattern matches the text.
3. If a match is found, return the starting index.
4. If no match is found after all slides, return -1.

#### Source Code (C++):

#include <iostream>

#include <string>

using namespace std;

int patternMatching(string text, string pattern) {

int n = text.length();

int m = pattern.length();

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

if (text.substr(i, m) == pattern) {

return i;

}

}

return -1;

}

int main() {

string text, pattern;

cout << "Enter the text: ";

cin >> text;

cout << "Enter the pattern: ";

cin >> pattern;

int result = patternMatching(text, pattern);

if (result == -1) {

cout << "Pattern not found!";

} else {

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

}

return 0;

}

#### Input:

#### Enter the text: ABABDABACDABABCABAB

#### Enter the pattern: ABABCABAB

#### 

#### Output:

#### Pattern found at index: 10

**Lab 6 : Write a program to implement a queue data structure along with its typical operations.**

#### Title:Implementing a Queue Data Structure with Typical Operations.

#### Theory:

A Queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. It supports two main operations: enqueue (insertion at the rear) and dequeue (deletion from the front).

#### Algorithm:

1. Define a queue using an array or linked list.
2. Implement the enqueue operation to add elements to the rear.
3. Implement the dequeue operation to remove elements from the front.
4. Implement additional operations like isEmpty, isFull, and peek.

#### 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;

}

**Output:**  
**Queue Operations:**

**1. Enqueue**

**2. Dequeue**

**3. Display**

**4. Exit**

**Enter your choice: 1**

**Enter the value to enqueue: 10**

**10 enqueued to the queue.**

**Current front: 0, rear: 0**

**Current queue: 10**

**Queue Operations:**

**1. Enqueue**

**2. Dequeue**

**3. Display**

**4. Exit**

**Enter your choice: 1**

**Enter the value to enqueue: 20**

**20 enqueued to the queue.**

**Current front: 0, rear: 1**

**Current queue: 10 20**

**Queue Operations:**

**1. Enqueue**

**2. Dequeue**

**3. Display**

**4. Exit**

**Enter your choice: 2**

**Dequeued value: 10**

**Current front: 1, rear: 1**

**Current queue: 20**

**Queue Operations:**

**1. Enqueue**

**2. Dequeue**

**3. Display**

**4. Exit**

**Enter your choice: 2**

**Dequeued value: 20**

**Current front: 2, rear: 1**

**Queue is now empty.**

**Queue is empty.**

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

#### Title: Solving the N-Queens Problem Using Backtracking.

#### Theory:

The N-Queens problem involves placing N queens on an N×N chessboard such that no two queens threaten each other. Backtracking is used to explore all possible configurations and find a valid solution.

#### Algorithm:

1. Start from the first row and try placing a queen in each column.
2. Check if the current placement is safe (no conflicts with previously placed queens).
3. If safe, move to the next row and repeat.
4. If not safe, backtrack and try the next column.
5. Repeat until all queens are placed or no solution is found.

#### Source Code (C++):

#include <iostream>

using namespace std;

#define N 4

void printSolution(int placed[])

{

static int solutionCount = 0;

cout << "\nSolution " << ++solutionCount << ":\n";

for (int i = 0; i < N; i++, cout << "\n")

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

cout << (placed[i] == j ? 'Q' : '.') << " ";

}

bool isSafe(int placed[], int row, int col)

{

for (int prev = 0; prev < row; prev++)

{

if (placed[prev] == col ||

placed[prev] - prev == col - row ||

placed[prev] + prev == col + row)

{

return false;

}

}

return true;

}

void solveNQueens(int placed[], int row)

{

if (row == N)

{

printSolution(placed);

return;

}

for (int col = 0; col < N; col++)

{

if (isSafe(placed, row, col))

{

placed[row] = col;

solveNQueens(placed, row + 1);

}

}

}

int main()

{

int placed[N] = {-1};

solveNQueens(placed, 0);

return 0;

}

#### Input: (N=4)

#### Output:

**Solution 1:**

**. Q . .**

**. . . Q**

**Q . . .**

**. . Q .**

**Solution 2:**

**. . Q .**

**Q . . .**

**. . . Q**

**. Q . .**

#### 

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

#### Title:Solving the Sum of Subsets Problem.

#### Theory:

The Sum of Subsets problem involves finding a subset of a given set of numbers that adds up to a specific target sum. Backtracking is used to explore all possible subsets.

#### Algorithm:

1. Start with an empty subset.
2. Add elements to the subset one by one.
3. If the current subset sum equals the target, print the subset.
4. If the sum exceeds the target, backtrack and try the next element.
5. Repeat until all subsets are explored.

#### Source Code (C++):

#include <iostream>

#include <vector>

using namespace std;

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

if (sum == target) {

for (int num : subset) {

cout << num << " ";

}

cout << endl;

return;

}

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

if (sum + set[i] <= target) {

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

sumOfSubsets(set, subset, sum + set[i], target, i + 1);

subset.pop\_back(); // Backtrack

}

}

}

int main() {

int n, target;

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

cin >> n;

vector<int> set(n);

cout << "Enter the elements: ";

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

cin >> set[i];

}

cout << "Enter the target sum: ";

cin >> target;

vector<int> subset;

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

return 0;

}

#### Input:

#### Enter the number of elements: 6

#### Enter the elements: 5 10 12 13 15 18

#### Enter the target sum: 30

#### 

#### Output:

#### 5 10 15

#### 5 12 13

#### 12 18

#### 

**Lab 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.**

**Title:Solving the 0/1 Knapsack Problem Using Dynamic Programming.**

#### Theory:

The 0/1 Knapsack problem is a classic optimization problem where the goal is to maximize the total value of items in a knapsack without exceeding its capacity. Each item can either be included (1) or not included (0). Dynamic Programming is used to solve this problem efficiently by breaking it down into smaller subproblems and storing their solutions in a table.

The time complexity is O(n⋅C), where

*n* is the number of items and

*C* is the knapsack capacity.

#### Algorithm:

1. Create a 2D DP table where rows represent items and columns represent capacities from 0 to C
2. Initialize the first row and column with 0 (no items or no capacity).
3. For each item, update the DP table:
   * If the item's weight is less than or equal to the current capacity, choose the maximum between including and excluding the item.
   * Otherwise, exclude the item.
4. The value in the last cell of the DP table represents the maximum profit.

#### 

#### Source Code (C++):

#### #include <iostream>

#### #include <vector>

#### using namespace std;

#### 

#### int knapsack(vector<int>& profits, vector<int>& weights, int capacity, int n) {

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

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

#### for (int j = 1; j <= capacity; j++) {

#### if (weights[i - 1] <= j) {

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

#### } else {

#### dp[i][j] = dp[i - 1][j];

#### }

#### }

#### }

#### return dp[n][capacity];

#### }

#### 

#### int main() {

#### int n, capacity;

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

#### cin >> n;

#### vector<int> profits(n), weights(n);

#### cout << "Enter the profits: ";

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

#### cin >> profits[i];

#### }

#### cout << "Enter the weights: ";

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

#### cin >> weights[i];

#### }

#### cout << "Enter the knapsack capacity: ";

#### cin >> capacity;

#### cout << "Maximum profit: " << knapsack(profits, weights, capacity, n);

#### return 0;

#### }

#### Input:

#### Enter the number of items: 4

#### Enter the profits: 15 25 13 23

#### Enter the weights: 2 6 12 9

#### Enter the knapsack capacity: 20

#### Output:

#### Maximum profit: 48

#### Lab 10 : Write a program to solve the Tower of Hanoi problem for the N disk. Title: Solving the Tower of Hanoi Problem for N Disks.

#### Theory:

The Tower of Hanoi is a mathematical puzzle that consists of three rods and N disks of different sizes. The objective is to move the entire stack from the starting rod to the destination rod, obeying the following 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 rod.  
   The problem can be solved using recursion, and the minimum number of moves required is .

#### Algorithm:

1. Move N−1 disks from the starting rod to the auxiliary rod.
2. Move the N*th* disk from the starting rod to the destination rod.
3. Move the *N*−1 disks from the auxiliary rod to the destination rod.

#### Source Code (C++):

#include <iostream>

using namespace std;

void towerOfHanoi(int n, char from, char to, char aux)

{

if (n == 1)

{

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

return;

}

towerOfHanoi(n - 1, from, aux, to);

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

towerOfHanoi(n - 1, aux, to, from);

}

int main()

{

int n;

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

cin >> n;

towerOfHanoi(n, 'A', 'C', 'B');

return 0;

}

#### Input: Enter the 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