|  |  |
| --- | --- |
|  | |
| Sl. | Problem description |
| 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* disc |

**Experiment no:1**

**Name:**

Write a program to sort a linear array using the bubble sort algorithm.

**Theory**:

Bubble Sort is a simple sorting algorithm that compares adjacent elements and swaps them if they are in the wrong order. This process repeats until the entire list is sorted.

Time Complexity:

Best case: O(n)

Worst/Average case: O(n²)

**Objective**:

Bubble Sort aims to sort an array by repeatedly comparing and swapping adjacent elements.

Time Complexity: O(n²) (worst/average), O(n) (best).

Space Complexity: O (1).

Stable: Equal elements retain their order.

Use: Best for small datasets or simplicity.

**Algorithm**:

1.Traverse the array.

2.Compare adjacent elements and swap them if needed.

3.After each iteration, the largest element is "bubbled" to its correct position.

4.Repeat the process until no swaps are needed.

**Code :**

#include <iostream>

using namespace std;

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

// Outer loop to traverse the array

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

bool swapped = false;

// Inner loop to perform comparisons and swapping

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

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

// Swap the elements if they are in the wrong order

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

swapped = true;

}

}

// If no two elements were swapped, the array is already sorted

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() {

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

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

cout << "Original Array: ";

printArray(arr, n);

bubbleSort(arr, n);

cout << "Sorted Array: ";

printArray(arr, n);

return 0;

}

Output:

Original Array: 64 34 25 12 22 11 90

Sorted Array: 11 12 22 25 34 64 90

**Experiment No:2**

**Name:** Write a program to find an element using a linear search algorithm.

**Theory:**

Linear Search is a basic search algorithm that checks each element of an array one by one to find a target element. If the element is found, its index is returned. If not, -1 is returned.

**Time Complexity:**

* Worst-case: O(n)
* Best-case: O(1)
* Average-case: O(n)

**Objective:**

To find an element in an array using the Linear Search algorithm.

**Algorithm:**

1. Loop through the array.
2. Compare each element with the target.
3. If a match is found, return the index.
4. If the loop finishes without finding the element, return -1.

**Code**:

#include <iostream>

using namespace std;

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

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

if (arr[i] == x) {

return i; // Return index if found

}

}

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

}

int main() {

int arr[] = {10, 20, 30, 40, 50};

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

int x = 30; // Element to search

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

if (result != -1)

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

else

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

return 0;

}

**Input:** arr = {10, 20, 30, 40, 50}, x = 30 **Output:** Element found at index 2

**Experiment No:3**

**Name :** Write a program to sort a linear array using the merge sort algorithm.

**Theory:**

Merge Sort is a divide-and-conquer algorithm. It recursively divides the array into two halves, sorts them, and then merges the sorted halves to produce a fully sorted array.

Time Complexity:

* Worst-case: O(n log n)
* Best-case: O(n log n)
* Average-case: O(n log n)

**Objective:**

To sort a linear array using the Merge Sort algorithm.

**Algorithm:**

1. Divide: Split the array into two halves.
2. Sort: Recursively sort each half.
3. Merge: Merge the sorted halves back into one sorted array.

**Code:**

#include <iostream>

using namespace std;

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

int n1 = mid - left + 1, n2 = right - mid;

int leftArr[n1], rightArr[n2];

for (int i = 0; i < n1; i++) leftArr[i] = arr[left + i];

for (int i = 0; i < n2; i++) rightArr[i] = arr[mid + 1 + i];

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

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

arr[k++] = (leftArr[i] <= rightArr[j]) ? leftArr[i++] : rightArr[j++];

}

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

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

}

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);

}

}

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

for (int i = 0; i < size; i++) cout << arr[i] << " ";

cout << endl;

}

int main() {

int arr[] = {12, 11, 13, 5, 6, 7};

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

cout << "Original Array: ";

printArray(arr, n);

mergeSort(arr, 0, n - 1);

cout << "Sorted Array: ";

printArray(arr, n);

return 0;

}

**Input:**

arr = {12, 11, 13, 5, 6, 7}

**Output:**

Original Array: 12 11 13 5 6 7

Sorted Array: 5 6 7 11 12 13

**Experiment No:4**

**Name:** Write a program to find an element using the binary search algorithm.

**Theory:**

Binary Search is a fast search algorithm that works on **sorted arrays** by repeatedly dividing the search space in half. It compares the target element with the middle element of the array and eliminates half of the elements based on the comparison.

**Time Complexity:**

* **Worst-case**: O(log n)
* **Best-case**: O(1) (if the middle element is the target)
* **Average-case**: O(log n)

**Objective:**

To find an element in a sorted array using the Binary Search algorithm.

**Algorithm (Binary Search):**

1. **Input**: A sorted array arr[] and the target element x.
2. **Output**: The index of the target element if found, or -1 if the element is not present.

Code:

#include <iostream>

using namespace std;

int binarySearch(int arr[], int n, int x) {

int left = 0, right = n - 1;

while (left <= right) {

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

if (arr[mid] == x) return mid;

else if (arr[mid] < x) left = mid + 1;

else right = mid - 1;

}

return -1;

}

int main() {

int arr[] = {1, 3, 5, 7, 9, 11};

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

int x = 7;

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

cout << (result != -1 ? "Element found at index " + to\_string(result) : "Element not found") << endl;

return 0;

}

**Input:**

arr = {1, 3, 5, 7, 9, 11}, x = 7

**Output:**

Element found at index 3

**Experiment no :5**

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

**Theory:**

Pattern matching is the process of finding a pattern (substring) within a text (string). The **Naive Pattern Matching Algorithm** checks every possible position in the text to see if the pattern matches, which is simple but not the most efficient.

**Objective:**

To find a given **pattern** in a **text** and return the starting index of the first occurrence of the pattern.

**Algorithm:**

1. Loop through the text and try to match the pattern starting from each position.
2. For each starting position, compare the characters of the pattern with the substring in the text.
3. If a match is found, return the index of the starting position.
4. If no match is found after checking all positions, return -1.

**Time Complexity:**

* **Worst-case**: O(n \* m), where n is the length of the text and m is the length of the pattern.

**Code:**

#include <iostream>

#include <string>

using namespace std;

// Naive Pattern Matching Algorithm

int naivePatternSearch(const string& text, const string& pattern) {

int n = text.length();

int m = pattern.length();

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

int j = 0;

while (j < m && text[i + j] == pattern[j]) {

j++;

}

if (j == m) return i; // Pattern found

}

return -1; // Pattern not found

}

int main() {

string text = "ABABABCABABABCABAB";

string pattern = "ABABC";

int result = naivePatternSearch(text, pattern);

if (result != -1)

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

else

cout << "Pattern not found" << endl;

return 0;

}

**Input:**

Text: "ABABABCABABABCABAB"

Pattern: "ABABC"

**Output:**

Pattern found at index: 4

**Experiment no: 6**

**Name:** 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 **First In, First Out (FIFO)** principle. The element that is added first is the one that gets removed first. Common operations are **enqueue** (add element), **dequeue** (remove element), and **peek** (view the front element).

**Objective:**

To implement a queue and perform its typical operations: **enqueue**, **dequeue**, **peek**, and check if the queue is empty or full.

**Algorithm:**

1. **Enqueue**: Add an element to the rear.
2. **Dequeue**: Remove an element from the front.
3. **Peek**: View the front element without removing it.
4. **isEmpty**: Check if the queue is empty.
5. **isFull**: Check if the queue is full.

#include <iostream>

using namespace std;

class Queue {

private:

int front, rear, size;

int \*arr;

public:

Queue(int n) {

size = n;

arr = new int[size];

front = -1;

rear = -1;

}

void enqueue(int value) {

if (rear == size - 1) cout << "Queue is full!" << endl;

else {

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

rear++;

arr[rear] = value;

}

}

int dequeue() {

if (front == -1 || front > rear) {

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

return -1;

}

return arr[front++];

}

int peek() {

if (front == -1 || front > rear) return -1;

return arr[front];

}

bool isEmpty() { return front == -1 || front > rear; }

bool isFull() { return rear == size - 1; }

void display() {

if (isEmpty()) cout << "Queue is empty!" << endl;

else {

for (int i = front; i <= rear; i++) cout << arr[i] << " ";

cout << endl;

}

}

};

int main() {

Queue q(5); // Queue size 5

q.enqueue(10); q.enqueue(20); q.enqueue(30);

q.display(); // 10 20 30

cout << "Dequeued: " << q.dequeue() << endl; // 10

q.display(); // 20 30

return 0;

}

**Output:**

10 20 30

Dequeued: 10 20 30

**Experiment No:7**

**Name:** Write a program to solve n queen's problem using backtracking.

**Theory:**

The **N-Queens problem** involves placing **N queens** on an **N×N** chessboard such that no two queens can attack each other. This is solved using **backtracking**, where queens are placed one by one, row by row, and the algorithm backtracks if a conflict arises, continuing until a valid solution is found.

**Objective:**

To place **N queens** on an **N×N** chessboard such that no two queens threaten each other (no two queens are in the same row, column, or diagonal). This is solved using the **backtracking** algorithm.

**Algorithm:**

1. **Start from the first row** and try placing a queen in each column.
2. **Check if the position is safe** (no queen in the same column or diagonals).
3. If safe, place the queen and move to the next row recursively.
4. If not safe, backtrack and try the next column in the current row.
5. If all rows are filled successfully, print the solution.
6. **Backtrack** by removing the queen and trying other options.

**Code:**

#include <iostream>

#include <vector>

using namespace std;

class NQueens {

private:

int N;

vector<vector<int>> board;

public:

NQueens(int n) {

N = n;

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

}

void printBoard() {

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

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

cout << (board[i][j] == 1 ? "Q" : ".") << " ";

}

cout << endl;

}

cout << endl;

}

bool isSafe(int row, int col) {

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

if (board[i][col] == 1) return false;

if (col - (row - i) >= 0 && board[i][col - (row - i)] == 1) return false;

if (col + (row - i) < N && board[i][col + (row - i)] == 1) return false;

}

return true;

}

bool solve(int row) {

if (row == N) {

printBoard();

return true;

}

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

if (isSafe(row, col)) {

board[row][col] = 1;

if (solve(row + 1)) return true;

board[row][col] = 0;

}

}

return false;

}

void solveNQueens() {

if (!solve(0)) cout << "No solution exists!" << endl;

}

};

int main() {

int N;

cout << "Enter the value of N: ";

cin >> N;

NQueens nq(N);

nq.solveNQueens();

return 0;

}

Output:

Q . . .

. . Q .

. Q . .

. . . Q

**Experiment No:8**

**Name:** 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 Subsets Problem** is a problem where we are given a set S of integers and a number d. The goal is to determine if there exists a subset of S whose sum is equal to d. This problem is typically solved using **backtracking** or **dynamic programming**.

**Objective:**

Given a set S = {5, 10, 12, 13, 15, 18} and a target sum d = 30, the objective is to find if there is a subset of S whose sum equals d.

**Algorithm (Backtracking):**

1. **Start with an empty subset** and a sum of 0.
2. **For each element** in the set, there are two choices:
   * **Include** the element in the current subset.
   * **Exclude** the element from the subset.
3. Recursively check both choices for each element and keep track of the sum.
4. If the sum equals d, print the subset.
5. Backtrack if the sum exceeds d or if no valid subset is found.

**Code:**

#include <iostream>

#include <vector>

using namespace std;

class SumOfSubsets {

private:

vector<int> set;

int targetSum;

vector<int> subset;

public:

SumOfSubsets(vector<int> s, int d) : set(s), targetSum(d) {}

// Recursive function to find subsets

void findSubset(int index, int currentSum) {

// If current sum matches the target sum, print the subset

if (currentSum == targetSum) {

cout << "Subset with sum " << targetSum << ": { ";

for (int num : subset) {

cout << num << " ";

}

cout << "}" << endl;

return;

}

// If all elements are considered, return

if (index == set.size()) return;

// Include the current element in the subset

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

findSubset(index + 1, currentSum + set[index]);

// Backtrack: exclude the current element from the subset

subset.pop\_back();

findSubset(index + 1, currentSum);

}

// Solve the problem

void solve() {

findSubset(0, 0);

}

};

int main() {

// Given set and target sum

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

int targetSum = 30;

SumOfSubsets solver(set, targetSum);

solver.solve();

return 0;

}

**Output:**

Subset with sum 30: {12 18}

Subset with sum 30: {5 10 15}

**Experiment No:9**

**Name:** 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.

**Theory:**

The **0/1 Knapsack Problem** is a classic problem in optimization. Given a set of items, each with a weight and a profit, the goal is to select a subset of these items to maximize the total profit while staying within a specified weight limit. In the **0/1 Knapsack**, each item can either be included (1) or excluded (0) from the knapsack.

**Objective:**

Given the profits and weights of items and a knapsack capacity, find the maximum profit that can be achieved without exceeding the knapsack's weight limit.

**Algorithm**

1. **Create a DP table**: dp[i][w] where i is the number of items and w is the capacity.
2. **Fill the table**:
   * For each item i and capacity w:
     + If item i can be included (weight[i] <= w), choose the max of:
       - Including the item: dp[i-1][w - weight[i]] + profit[i]
       - Excluding the item: dp[i-1][w]
3. **Return the result**: The maximum profit is dp[n][C], where n is the number of items and C is the knapsack capacity.

**Code:**

#include <iostream>

#include <vector>

using namespace std;

class Knapsack {

private:

vector<int> profits, weights;

int capacity, n;

public:

Knapsack(vector<int> p, vector<int> w, int c, int numItems) : profits(p), weights(w), capacity(c), n(numItems) {}

int knapsackDP() {

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

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

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

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

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

} else {

dp[i][w] = dp[i - 1][w];

}

}

}

return dp[n][capacity];

}

};

int main() {

vector<int> profits = {15, 25, 13, 23};

vector<int> weights = {2, 6, 12, 9};

int capacity = 20, n = 4;

Knapsack knapsack(profits, weights, capacity, n);

cout << "Max profit: " << knapsack.knapsackDP() << endl;

return 0;

}

**Output:**

Max profit: 38

**Experiment No:10**

**Name:** Write a program to solve the Tower of Hanoi problem for the N disk.

**Theory:**

The **Tower of Hanoi** is a classic puzzle in recursion, involving three pegs and a number of disks of different sizes. The objective is to move all the disks from the first peg to the third peg, following these rules:

1. Only one disk can be moved at a time.
2. A disk can only be moved to the top of another peg if it is smaller than the disk already on that peg.
3. No disk may be placed on top of a smaller disk.

**Objective:**

Move N disks from the source peg to the destination peg using an auxiliary 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.

**Algorithm (Recursive):**

1. **Base case**: If there is one disk, move it directly from the source to the destination peg.
2. **Recursive case**:
   * Move the top N-1 disks from the source to the auxiliary peg.
   * Move the Nth (largest) disk from the source to the destination peg.
   * Move the N-1 disks from the auxiliary peg to the destination peg.

**Code:**

#include <iostream>

using namespace std;

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

if (n == 1) {

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

return;

}

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

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

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

}

int main() {

int n;

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

cin >> n;

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

return 0;

}

**Output:**

Enter the number of disks: 3

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