**Pabna University of Science and Technology**



**Lab Report**

**Information and Communication Engineering**

**Course Code : Data Structure and Algorithm Sessional.**

**Course Title:ICE-2202**

**Submitted By: Submitted To:**

**Name :Sumaiya Hossain Md. Anwar Hossain**

**Roll: 220633 Professor**

**Session:2021-2022 Department of ICE,PUST**

**Department of ICE,PUST**

**Date of Submission :28-02-2025**

**Index**

|  |  |
| --- | --- |
| **Sl.** | **Problem Statement** |
| **1.** | |  | | --- | | Write a program to sort a linear array using the bubble sort algorithm. | |
| **2.** | |  | | --- | | Write a program to find an element using a linear search algorithm. | |
| **3.** | |  | | --- | | Write a program to sort a linear array using the merge sort algorithm. | |
| **4.** | |  | | --- | | Write a program to find an element using the binary search algorithm. | |
| **5.** | |  | | --- | | Write a program to find a given pattern from text using the pattern matching algorithm. | |
| **6.** | |  | | --- | | Write a program to implement a queue data structure along with its typical operations. | |
| **7.** | |  | | --- | | Write a program to solve **n** queen's problem using backtracking. | |
| **8.** | |  | | --- | | Consider a set **S = {5,10,12,13,15,18}** and **d = 30**. Write a program to solve the sum of subset problem. | |
| **9.** | |  | | --- | | Write a program to solve the following **0/1 Knapsack** using dynamic programming approach **profits P = (15,25,13,23), weight W = (2,6,12,9), Knapsack C = 20**, and the number of items **n=4**. | |
| **10** | |  | | --- | | Write a program to solve the **Tower of Hanoi** problem for the **N** disk. | |

**Index**

**Experiment No:01**

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

**Theory:**

Bubble sort is one of the simplest sorting algorithms. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted. It's called "bubble sort" because smaller elements "bubble" to the top of the list.

**Algorithm:**

1. Start at the beginning of the list.

2. Compare the first two elements.

3. If the first element is greater than the second element, swap them.

4. Move to the next pair of elements and repeat step 3.

5. Continue this process for each pair of adjacent elements to the end of the list.

6. After each pass, the next largest element will be in its correct position.

7. Repeat the process for the remaining unsorted elements.

**Source Code:**

#include <iostream>

using namespace std;

// Function to perform bubble sort

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

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

// Last i elements are already in place

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

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

// Swap arr[j] and arr[j+1]

int temp = arr[j];

arr[j] = arr[j+1];

arr[j+1] = temp;

}

}

}

}

// Function to print the array

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

for (int i = 0; i < size; 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 << "Unsorted array: ";

printArray(arr, n);

bubbleSort(arr, n);

cout << "Sorted array: ";

printArray(arr, n);

return 0;

}

**Input:**

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

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

**Output:**

Unsorted array: 64 34 25 12 22 11 90

Sorted array: 11 12 22 25 34 64 90

**Experiment No:02**

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

**Theory:**

Linear search, also known as sequential search, is a straightforward search algorithm. It works by iterating through each element of the list sequentially until the desired element is found or the list ends. This algorithm is simple and does not require the list to be sorted.

**Algorithm:**

1. Start from the leftmost element of the list.

2. Compare the target element with the current element.

3. If the target element matches the current element, return the current index.

4. If the target element does not match the current element, move to the next element in the list.

5. Repeat steps 2-4 until the element is found or the end of the list is reached.

6. If the element is not found, return a value indicating that the element is not present (e.g., -1).

**Source Code:**

#include <iostream>

using namespace std;

// Function to perform linear search

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

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

if (arr[i] == target) {

return i; // Element found, return index

}

}

return -1; // Element not found, return -1

}

// Function to print the result

void printResult(int index) {

if (index != -1) {

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

} else {

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

}

}

int main() {

int arr[] = {10, 23, 45, 78, 12, 34};

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

int target = 78; // Element to be searched

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

printResult(result);

return 0;

}

**Input:**

int arr[] = {10, 23, 45, 78, 12, 34};

int target = 78;

**Output:**

Element found at index: 3

**Experiment No:03**

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

**Theory:**

Merge sort is an efficient, stable, and comparison-based sorting algorithm. It's based on the divide-and-conquer paradigm and recursively divides the array into two halves until each subarray has only one element. It then merges the subarrays in a sorted manner.

**Algorithm:**

1. Divide the unsorted array into two approximately equal halves.

2. Recursively sort each half using merge sort.

3. Merge the two sorted halves to produce a single sorted array.

4. Continue the process until the entire array is sorted.

**Source Code:**

#include <iostream>

using namespace std;

// Function to merge two halves

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

int n1 = mid - left + 1;

int n2 = right - mid;

int L[n1], R[n2];

for (int i = 0; i < n1; i++)

L[i] = arr[left + i];

for (int j = 0; j < n2; j++)

R[j] = arr[mid + 1 + j];

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

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

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[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;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

// Function to print the array

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

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

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

cout << endl;

}

int main() {

int arr[] = {38, 27, 43, 3, 9, 82, 10};

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

cout << "Unsorted array: ";

printArray(arr, n);

mergeSort(arr, 0, n - 1);

cout << "Sorted array: ";

printArray(arr, n);

return 0;

}

**Input:**

int arr[] = {38, 27, 43, 3, 9, 82, 10};

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

**Output:**

Unsorted array: 38 27 43 3 9 82 10

Sorted array: 3 9 10 27 38 43 82

**Experiment No:04**

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

**Theory:**

Binary search is an efficient algorithm for finding an element in a sorted array. It works by repeatedly dividing the search interval in half. If the value of the search key is less than the item in the middle of the interval, the interval is narrowed to the lower half; otherwise, it is narrowed to the upper half. The process continues until the value is found or the interval is empty.

**Algorithm:**

1. Start with two pointers, one at the low end (left) and one at the high end (right) of the array.

2. Calculate the middle index: mid = left + (right - left) / 2.

3. Compare the middle element with the target element:

o If the middle element is equal to the target, return the middle index.

o If the middle element is greater than the target, narrow the search interval to the lower half.

o If the middle element is less than the target, narrow the search interval to the upper half.

4. Repeat steps 2-3 until the target element is found or the interval is empty.

5. If the target element is not found, return a value indicating that the element is not present (e.g., -1).

**Source Code:**

#include <iostream>

using namespace std;

// Function to perform binary search

int binarySearch(int arr[], int left, int right, int target) {

while (left <= right) {

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

// Check if the target is present at mid

if (arr[mid] == target)

return mid;

// If target is greater, ignore the left half

if (arr[mid] < target)

left = mid + 1;

else

// If target is smaller, ignore the right half

right = mid - 1;

}

// Target is not present in the array

return -1;

}

// Function to print the result

void printResult(int index) {

if (index != -1) {

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

} else {

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

}

}

int main() {

int arr[] = {2, 3, 4, 10, 40};

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

int target = 10; // Element to be searched

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

printResult(result);

return 0;

}

**Input:**

int arr[] = {2, 3, 4, 10, 40};

int target = 10;

**Output:**

Element found at index: 3

**Experiment No:05**

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

**Theory:**

The KMP algorithm improves on the naive pattern matching approach by avoiding unnecessary comparisons. It preprocesses the pattern to create a partial match table (also known as the "longest proper prefix which is also a suffix" or LPS array). This table is used to skip sections of the text where mismatches are found, thus reducing the number of comparisons.

**Algorithm:**

1. Preprocessing the Pattern:

o Construct the LPS array for the given pattern.

o The LPS array stores the length of the longest proper prefix of the pattern that is also a suffix for each subpattern.

2. Searching the Pattern:

o Initialize pointers for the text and the pattern.

o Compare characters of the text and the pattern.

o If a match is found, move to the next characters of both text and pattern.

o If a mismatch is found, use the LPS array to shift the pattern to the right without re-examining previously matched characters.

o Repeat until the pattern is found or the text is exhausted.

**Source Code:**

#include <iostream>

#include <vector>

#include <string>

using namespace std;

// Function to construct the LPS array

void computeLPSArray(const string& pattern, vector<int>& lps) {

int length = 0;

lps[0] = 0; // lps[0] is always 0

int i = 1;

while (i < pattern.length()) {

if (pattern[i] == pattern[length]) {

length++;

lps[i] = length;

i++;

} else {

if (length != 0) {

length = lps[length - 1];

} else {

lps[i] = 0;

i++;

}

}

}

}

// Function to perform KMP pattern matching

void KMPsearch(const string& text, const string& pattern) {

int M = pattern.length();

int N = text.length();

vector<int> lps(M);

computeLPSArray(pattern, lps);

int i = 0; // index for text[]

int j = 0; // index for pattern[]

while (i < N) {

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

i++;

j++;

}

if (j == M) {

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

j = lps[j - 1];

} else if (i < N && pattern[j] != text[i]) {

if (j != 0) {

j = lps[j - 1];

} else {

i++;

}

}

}

}

int main() {

string text = "ABABDABACDABABCABAB";

string pattern = "ABABCABAB";

KMPsearch(text, pattern);

return 0;

}

**Input:**

string text = "ABABDABACDABABCABAB";

string pattern = "ABABCABAB";

**Output:**

Pattern found at index 10

**Experiment No:06**

**Experiment 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. Elements are added to the rear (enqueue operation) and removed from the front (dequeue operation). Think of it as a line at a ticket counter: the first person to get in line is the first person to get a ticket.

Typical Operations:

1. Enqueue: Add an element to the rear of the queue.

2. Dequeue: Remove an element from the front of the queue.

3. Front: Get the front element without removing it.

4. IsEmpty: Check if the queue is empty.

5. Size: Get the number of elements in the queue.

**Algorithm:**

1. Enqueue:

o If the queue is not full, add the element to the rear.

2. Dequeue:

o If the queue is not empty, remove the front element.

3. Front:

o If the queue is not empty, return the front element.

4. IsEmpty:

o Return true if the queue is empty; otherwise, return false.

5. Size:

o Return the number of elements in the queue.

**Source Code:**

#include <iostream>

using namespace std;

#define MAX 1000

class Queue {

int front, rear, size;

int arr[MAX];

public:

Queue() {

front = size = 0;

rear = MAX - 1;

}

bool isFull() {

return (size == MAX);

}

bool isEmpty() {

return (size == 0);

}

void enqueue(int item) {

if (isFull())

return;

rear = (rear + 1) % MAX;

arr[rear] = item;

size++;

cout << item << " enqueued to queue\n";

}

int dequeue() {

if (isEmpty())

return -1;

int item = arr[front];

front = (front + 1) % MAX;

size--;

return item;

}

int getFront() {

if (isEmpty())

return -1;

return arr[front];

}

int getSize() {

return size;

}

};

int main() {

Queue q;

q.enqueue(10);

q.enqueue(20);

q.enqueue(30);

q.enqueue(40);

cout << "Dequeued from queue: " << q.dequeue() << endl;

cout << "Front item is: " << q.getFront() << endl;

cout << "Queue size is: " << q.getSize() << endl;

return 0;

}

**Input:**

q.enqueue(10);

q.enqueue(20);

q.enqueue(30);

q.enqueue(40);

**Output:**

10 enqueued to queue

20 enqueued to queue

30 enqueued to queue

40 enqueued to queue

Dequeued from queue: 10

Front item is: 20

Queue size is: 3

**Experiment No:07**

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

**Theory:**

The N-Queens problem involves placing N chess queens on an N×N chessboard so that no two queens threaten each other. This means no two queens can be in the same row, column, or diagonal. The backtracking approach incrementally builds a solution, abandoning partial solutions (backtracks) as soon as it determines that the solution cannot possibly be completed to a valid solution.

**Algorithm:**

1. Start in the leftmost column.

2. Place the queen in the current column and try to place it in a safe row.

3. If placing the queen in the current row and column does not lead to a solution, backtrack and move the queen to the next row.

4. If placing the queen in the current column and row leads to a solution, move to the next column and repeat the process.

5. If all queens are placed in the correct position, return the solution.

6. If no safe position is found, return false to indicate no solution exists.

**Source Code:**

#include <iostream>

#include <vector>

using namespace std;

// Function to print the board

void printBoard(const vector<vector<int>>& board) {

for (const auto& row : board) {

for (int col : row) {

cout << (col ? "Q " : ". ");

}

cout << endl;

}

}

// Function to check if a queen can be placed at board[row][col]

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

// Check this row on left side

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

if (board[row][i]) {

return false;

}

}

// Check upper diagonal on left side

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

if (board[i][j]) {

return false;

}

}

// Check lower diagonal on left side

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

if (board[i][j]) {

return false;

}

}

return true;

}

// Function to solve N-Queens problem using backtracking

bool solveNQueens(vector<vector<int>>& board, int col, int n) {

if (col >= n) {

return true;

}

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

if (isSafe(board, i, col, n)) {

board[i][col] = 1;

if (solveNQueens(board, col + 1, n)) {

return true;

}

board[i][col] = 0; // backtrack

}

}

return false;

}

int main() {

int n = 4;

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

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

printBoard(board);

} else {

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

}

return 0;

}

**Input:**

Set the board size to 4 (N = 4):

int n = 4;

**Output:**

. Q . .

. . . Q

Q . . .

. . Q .

**Experiment No:08**

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

**Theory:**

The subset-sum problem is a classic problem in computer science, particularly in the field of combinatorial optimization. Given a set of non-negative integers and a value d, the task is to determine if there exists a subset whose sum is equal to d.

**Algorithm:**

We can solve this problem using backtracking. The approach involves:

1. Checking if including an element in the subset helps to reach the target sum.

2. If not, excluding the element and trying the next one.

3. The base case is reached when the target sum is zero (a valid subset is found) or when the set is exhausted without reaching the target sum.

**Source Code:**

#include <iostream>

#include <vector>

using namespace std;

// Function to print the subset

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

for (int num : subset) {

cout << num << " ";

}

cout << endl;

}

// Function to find subsets that sum up to target

bool isSubsetSum(const vector<int>& S, int n, int target, vector<int>& subset) {

if (target == 0) {

printSubset(subset);

return true;

}

if (n == 0 && target != 0) {

return false;

}

// Exclude the last element and recur

bool exclude = isSubsetSum(S, n - 1, target, subset);

// Include the last element and recur

subset.push\_back(S[n - 1]);

bool include = isSubsetSum(S, n - 1, target - S[n - 1], subset);

subset.pop\_back();

return exclude || include;

}

int main() {

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

int target = 10;

vector<int> subset;

if (!isSubsetSum(S, S.size(), target, subset)) {

cout << "No subset with sum " << target << " found." << endl;

}

return 0;

}

**Input:**

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

int target = 10;

**Output:**

5 5

5 10

No subset with sum 10 found.

**Experiment No:09**

**Experiment 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 involves a knapsack with a weight capacity C and a set of items, each with a weight W and a profit P. The goal is to maximize the total profit while ensuring that the total weight does not exceed the knapsack's capacity. The "0/1" aspect means that each item can either be included (1) or not included (0) in the knapsack.

**Algorithm:**

1. Create a 2D array dp of dimensions (n + 1) x (C + 1), where n is the number of items and C is the knapsack capacity. The value dp[i][j] will represent the maximum profit that can be achieved with the first i items and a knapsack capacity of j.

2. Initialize the first row and first column of the array with zeros, as no profit can be achieved with zero items or zero capacity.

3. Iterate through each item (from 1 to n) and each capacity (from 1 to C). For each combination, decide whether to include the current item or not:

o If the item's weight is less than or equal to the current capacity, choose the maximum between including the item (adding its profit and reducing the capacity) or not including it.

o Otherwise, do not include the item.

4. The value at dp[n][C] will represent the maximum profit that can be achieved with the given items and knapsack capacity.

**Source Code:**

#include <iostream>

#include <vector>

using namespace std;

// Function to solve the 0/1 Knapsack problem

int knapsack(const vector<int>& profits, const 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 w = 1; w <= capacity; w++) {

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

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

} else {

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

}

}

}

return dp[n][capacity];

}

int main() {

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

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

int capacity = 20;

int n = profits.size();

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

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

return 0;

}

**Input:**

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

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

int capacity = 20;

**Output:**

Maximum profit: 48

**Experiment No:10**

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

**Theory:**

The Tower of Hanoi is a mathematical puzzle that consists of three rods and NN disks of different sizes. The goal is to move all the disks from the source rod to the target rod using an auxiliary rod, 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 rod.

3. All disks start on the source rod and must be moved to the target rod.

**Algorithm:**

The problem can be solved using a recursive algorithm. The idea is to break down the problem into smaller sub-problems until the simplest case is reached (moving a single disk).

1. Base Case: If there is only one disk (N=1N = 1), move it directly from the source rod to the target rod.

2. Recursive Case: If there are more than one disk (N>1N > 1):

o Move the top N−1N-1 disks from the source rod to the auxiliary rod, using the target rod as an auxiliary.

o Move the NNth (largest) disk from the source rod to the target rod.

o Move the N−1N-1 disks from the auxiliary rod to the target rod, using the source rod as an auxiliary.

**Source Code:**

#include <iostream>

using namespace std;

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

if (n == 1) {

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

return;

}

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

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

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

}

int main() {

int N = 3; // Number of disks

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

return 0;

}

**Input:**

int N = 3; // Number of disks

**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