PABNA UNIVERSITY OF SCIENCE AND TECHNOLOGY

Department of Information and Comunication Engineering

LAB REPORT

DSA-2202 DATA STRUCTURE AND ALGORITHM SESSIONAL

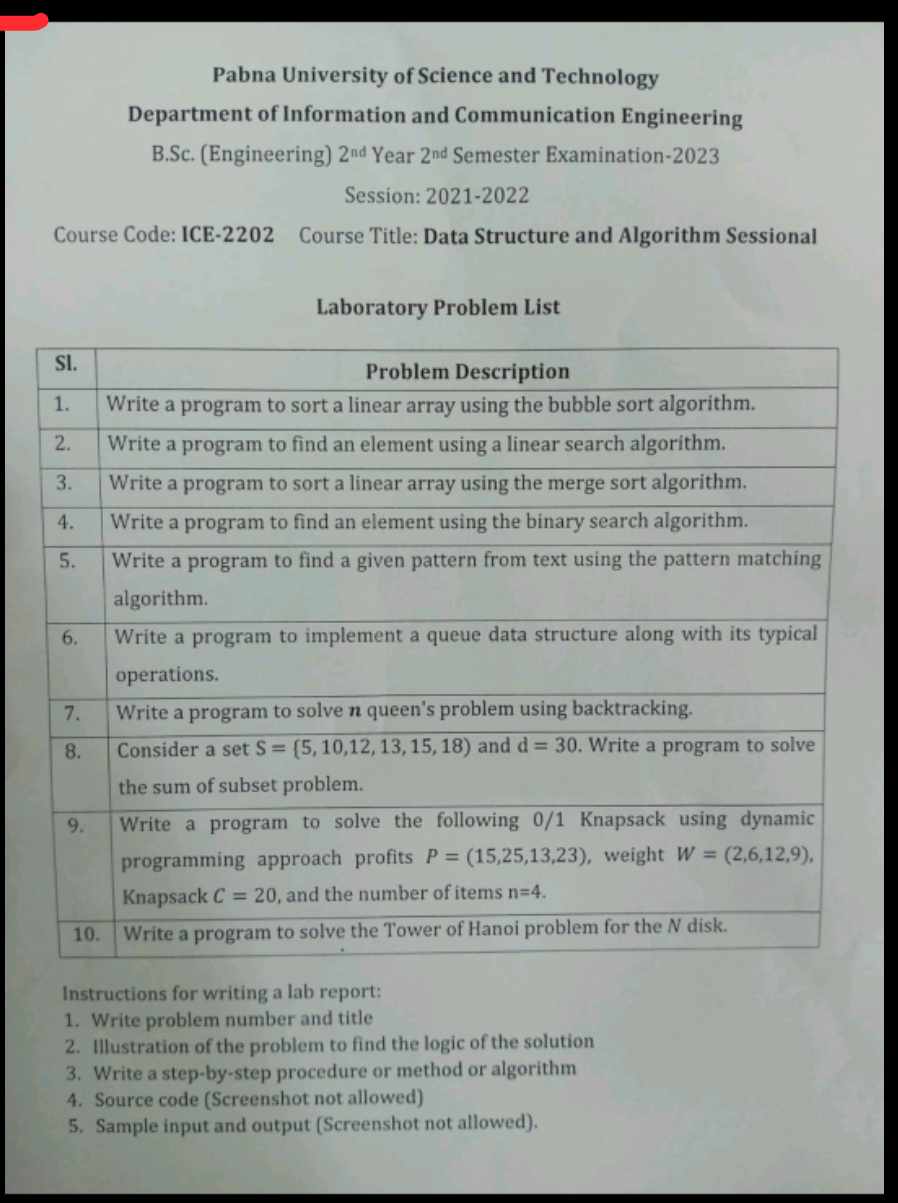
Submitted by submitted to

Name:Sayara Tasnim Shipra Md.Anowar Hossain

Roll:220631 professor

Dept.of Information and Department of Information and

Comunication Engineering communication Engineering



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

Ans:

Tittle: write a c program to sort a linear array using the bubble sort algorithm

Theory: Bubble Sort is a simple comparison-based sorting algorithm. The core idea is to repeatedly step through the list, compare adjacent elements, and swap them if they are in the wrong order. The process is repeated until the list is sorted. It is called "Bubble Sort" because smaller elements "bubble" to the top (beginning of the array) while larger elements "sink" to the bottom (end of the array).

Algorithm:

1. Starting from the first element of the array, compare the current element with the next element.

2.Swap the elements if the current element is greater than the next element.

3.Repeat the comparison for every pair of adjacent elements from the beginning to the end of the array.

4.After each complete pass through the array, the largest element will be placed in its correct position at the end.

5.Repeat the process for the rest of the elements (excluding the last sorted element in each iteration).

6.Stop when no swaps are made in a complete pass, which indicates the array is sorted.

Sourse code:

#include <stdio.h>

// Function to perform bubble sort on an array

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

int temp;

for (int i = 0; i < n-1; i++) { // Outer loop for number of passes

for (int j = 0; j < n-i-1; j++) { // Inner loop for each pass

// If the current element is greater than the next, swap them

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

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++) {

printf("%d ", arr[i]);

}

printf("\n");

}

int main() {

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

int n = sizeof(arr) / sizeof(arr[0]); // Find the size of the array

printf("Original array: \n");

printArray(arr, n);

bubbleSort(arr, n);

printf("\nSorted array: \n");

printArray(arr, n);11

return 0;

}

Input:64,34,25,12,11,90

Output:11,12,25,34,64,90

2.write a c program to find an element using a linear search algorithm.

Tittle: write a c program to find an element using a linear search algorithm.

Theory: Linear Search is a simple search algorithm used to find the position of a target value within a list. It checks each element of the array sequentially until a match is found or all elements have been checked. It is most effective on unsorted or small datasets.

Algorithm:

1.Start from the first element of the array.

2.Compare the current element with the target value.

3.If the current element matches the target value, return the index of that element.

4.If the current element does not match, move to the next element and repeat the comparison.

5.If the target element is not found after checking all elements, return -1 (indicating the target is not in the list).

Source code:

#include <stdio.h>

// 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; // Return the index if the element is found

}

}

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

}

int main() {

int arr[] = {12, 34, 55, 2, 10, 6, 89};

int n = sizeof(arr) / sizeof(arr[0]); // Find the size of the array

int target;

printf("Enter the element to search for: ");

scanf("%d", &target);

// Perform linear search

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

// Output result

if (result != -1) {

printf("Element %d found at index %d.\n", target, result);

} else {

printf("Element %d not found in the array.\n", target);

}

return 0;

}

Input: Enter the element to search: 10

Output:Element 10 found at index 4.

3.write a c program to sort a linear array using the merge sort algorithm

Tittle: write a c program to sort a linear array using the merge sort algorithm

Theory: Divide:

Split the unsorted array into two halves. If the array has only one element, it is already sorted.

Conquer:

Recursively sort both halves of the array.

Combine:

Merge the two sorted halves to produce a fully sorted array.

The merging process works by comparing the elements of the two halves and arranging them in order. This process continues until all elements from both halves are merged into a single sorted array.

Algorithm:

Source code: #include <stdio.h>

// Function to merge two subarrays into a sorted 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

// Temporary arrays to hold the left and right subarrays

int L[n1], R[n2];

// Copy data to temporary arrays L[] and R[]

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];

// Merge the temporary arrays back into arr[]

int i = 0; // Initial index of left subarray

int j = 0; // Initial index of right subarray

int k = left; // Initial index of merged subarray

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

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

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

// Copy the remaining elements of L[], if any

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

// Copy the remaining elements of R[], if any

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

// Function to implement merge sort

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

if (left < right) {

// Find the middle point

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

// Sort the first half

mergeSort(arr, left, mid);

// Sort the second half

mergeSort(arr, mid + 1, right);

// Merge the two halves

merge(arr, left, mid, right);

}

}

// Function to print the array

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

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

printf("%d ", arr[i]);

printf("\n");

}

int main() {

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

int arr\_size = sizeof(arr) / sizeof(arr[0]);

printf("Given array: \n");

printArray(arr, arr\_size);

mergeSort(arr, 0, arr\_size - 1);

printf("\nSorted array: \n");

printArray(arr, arr\_size);

return 0;

}

Input: 12 11 13 5 6 7

Output:

Sorted array:

5 6 7 11 12 13

4.write a c program to find an element using the binary search algorithm

Tittle: write a c program to find an element using the binary search algorithm

Theory: Binary Search is an efficient algorithm used to find the position of a target value within a sorted array. The basic idea is to repeatedly divide the search interval in half. If the value of the target is less than the value in the middle of the interval, the search continues on the left half. If the target is greater, it continues on the right half. This process is repeated until the target value is found or the interval is empty.

Algorithm:1. Initial Setup: The array must be sorted.

2.Middle Element: Find the middle element of the array.

3.Compare:

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

I f the target value is less than the middle element, repeat the search on the left half of the array.

If the target value is greater than the middle element, repeat the search on the right half of the array.

4.Termination: If the target is found, return the index. If the search range is empty (i.e., the left index exceeds the right index), the target is not in the array.

Source code: #include <stdio.h>

// Function to perform binary search

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

int left = 0, right = size - 1;

while (left <= right) {

int mid = left + (right - left) / 2; // Find the middle index

// Check if target is at the middle

if (arr[mid] == target) {

return mid; // Target found, return index

}

// If target is greater, ignore the left half

if (arr[mid] < target) {

left = mid + 1;

}

// If target is smaller, ignore the right half

else {

right = mid - 1;

}

}

// If the element is not present

return -1;

}

// Function to print the array

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

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

printf("%d ", arr[i]);

}

printf("\n");

}

int main() {

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

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

int target = 10;

printf("Given array: \n");

printArray(arr, size);

// Perform binary search

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

if (result != -1) {

printf("Element %d is present at index %d.\n", target, result);

} else {

printf("Element %d is not present in the array.\n", target);

}

return 0;

}

Input: Given array:

2 3 4 10 40

Output:

Element 10 is present at index 3.

5. Write a program to find a given pattern from txet using the pattern matching algorithm

Title: Write a program to find a given pattern from txet using the pattern matching algorithm .

Theory: In the Naive String Matching algorithm, the idea is to check for the given pattern at every possible position in the text. This algorithm slides the pattern over the text one by one, and for each position, it checks whether the pattern matches the substring starting at that position.

Algorithm: 1.Start from the beginning of the text.

2.For each position, check if the pattern matches the substring starting from that position.

3.If a match is found, return the position (index).

4.If the end of the text is reached, and no match is found, conclude that the pattern does not exist in the text.

Sourse code: #include <stdio.h>

#include <string.h>

// Function to perform Naive String Matching

void naive\_pattern\_matching(char \*text, char \*pattern) {

int n = strlen(text); // Length of the text

int m = strlen(pattern); // Length of the pattern

// Traverse through the text

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

// Check if the substring starting from i matches the pattern

int j = 0;

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

j++;

}

// If the whole pattern is matched, print the index

if (j == m) {

printf("Pattern found at index %d\n", i);

}

}

}

int main() {

char text[] = "THIS IS A SIMPLE EXAMPLE";

char pattern[] = "SIMPLE";

naive\_pattern\_matching(text, pattern);

return 0;

}

Input: Enter the text: THIS IS A SIMPLE EXAMPLE

Enter the pattern: SIMPLE

Output: Pattern found at index 10

6. Write a program to find implement a queue data stucture along with its typical operation .

Title: Write a program to find implement a queue data stucture along with its typical operation

Theory: A Queue is a linear data structure that follows the First In, First Out (FIFO) principle. This means that the element added first will be the one that gets removed first, like a queue in a bank or a line at a ticket counter.

Algorithm: InitializeQueue

Initialize an empty queue with front = -1 and rear = -1.

2. Enqueue(value)

If queue is full (rear == MAX - 1), print "Queue is full!".

If queue is not full, insert value at the rear and update pointers (front and rear).

3. Dequeue()

If the queue is empty (front == -1), print "Queue is empty!".

If queue is not empty, remove the element at front, update pointers (front and rear).

4. Front()

If the queue is empty (front == -1), print "Queue is empty!".

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

5. Rear()

If the queue is empty (front == -1), print "Queue is empty!".

If queue is not empty, return the element at rear.

6. isEmpty()

If front == -1, return True.

Else, return False.

7. isFull()

If rear == MAX - 1, return True.

Else, return False.

Source code:

#include <stdio.h>

#define MAX 5

int queue[MAX], front = -1, rear = -1;

// Check if queue is empty

int isEmpty() {

return front == -1;

}

// Check if queue is full

int isFull() {

return rear == MAX - 1;

}

// Enqueue operation

void enqueue(int value) {

if (isFull()) {

printf("Queue is full!\n");

} else {

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

queue[++rear] = value;

printf("Enqueued: %d\n", value);

}

}

// Dequeue operation

int dequeue() {

if (isEmpty()) {

printf("Queue is empty!\n");

return -1;

} else {

int value = queue[front];

if (front == rear) front = rear = -1; // Queue becomes empty

else front++;

return value;

}

}

// Front operation

int getFront() {

return isEmpty() ? -1 : queue[front];

}

// Rear operation

int getRear() {

return isEmpty() ? -1 : queue[rear];

}

int main() {

enqueue(10); enqueue(20); enqueue(30);

printf("Front: %d, Rear: %d\n", getFront(), getRear());

printf("Dequeued: %d\n", dequeue());

enqueue(40); enqueue(50); enqueue(60);

printf("Front: %d, Rear: %d\n", getFront(), getRear());

printf("Dequeued: %d\n", dequeue());

return 0;

}

7. Write a program to solve n queens problem using backtracking

Title: Write a program to solve n queens problem using backtracking

Theory:

N-Queens Problem:

The N-Queens problem is a classic problem in computer science, where the task is to place N queens on an N×N chessboard such that no two queens threaten each other. That is, no two queens can share the same row, column, or diagonal.

Backtracking Approach:

The backtracking approach is a systematic way of trying different possibilities for solving a problem. It works by placing queens on the board and checking if the current configuration is valid. If placing a queen in the current position leads to a solution, it proceeds to the next row. If placing a queen leads to an invalid state (a conflict with previously placed queens), it backtracks and tries the next possible position.

Algorithm: 1.Start from the first row.

2.Try placing a queen in every column of the current row.

3.Check if the placement is valid: A queen is valid if no other queens are in the same column or diagonals.

4.If placing the queen is valid, move to the next row and repeat the process.

5.If all queens are placed successfully, print the solution.

6.If placing the queen leads to a conflict, backtrack and move the queen to the next column in the previous row.

Source code:

#include <stdio.h>

#define MAX 10

int board[MAX][MAX];

// Function to print the board

void printBoard(int n) {

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

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

printf(board[i][j] ? " Q " : " . ");

printf("\n");

}

}

// Check if it's safe to place a queen

int isSafe(int n, int row, int col) {

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

if (board[i][col] || (col - (row - i) >= 0 && board[i][col - (row - i)]) || (col + (row - i) < n && board[i][col + (row - i)])) {

return 0;

}

}

return 1;

}

// Solve using backtracking

int solveNQueens(int n, int row) {

if (row == n) return 1;

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

if (isSafe(n, row, col)) {

board[row][col] = 1;

if (solveNQueens(n, row + 1)) return 1;

board[row][col] = 0; // Backtrack

}

}

return 0;

}

int main() {

int n;

printf("Enter the number of queens: ");

scanf("%d", &n);

if (solveNQueens(n, 0)) {

printBoard(n);

} else {

printf("No solution exists for %d queens.\n", n);

}

return 0;

}

Input: Enter the number of queens: 4

Output: Q . . .

. . Q .

. Q . .

. . . Q

8. consider a set S={5,10,12,13,15,18}and d=30 write a c program to solve the sum pof subset problem

Title: consider a set S={5,10,12,13,15,18}and d=30 write a c program to solve the sum pof subset problem

Theory: The subset sum problem is a classical problem in computer science and falls under the category of combinatorial optimization problems. It asks whether a subset of a given set of integers can sum up to a given number

𝑑

d.

For this problem, the set

𝑆

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

S={5,10,12,13,15,18} and the target sum is 30.

The core idea of solving this problem is to check all possible subsets of

𝑆

S. This can be done using a dynamic programming approach, which reduces the computational complexity. Instead of checking every subset explicitly (which would be very time-consuming), dynamic programming leverages previously computed results to build up the solution efficiently.

Algorithm: We create a 2D table dp, where dp[i][j] represents whether it is possible to form the sum j using the first i elements of the set.

Initialize the table with:

dp[i][0] = true for all

𝑖

i, because a sum of 0 can always be formed with an empty subset.

dp[0][j] = false for all

𝑗

>

0

j>0, because no sum greater than 0 can be formed with an empty set.

Then, for each element in the set, update the table to reflect whether including the current element can form the target sum.

Source code: #include <stdio.h>

#include <stdbool.h>

#define MAX\_SIZE 6 // Size of the set S

#define TARGET 30 // Target sum

// Function to solve the subset sum problem using dynamic programming

bool isSubsetSum(int set[], int n, int target) {

bool dp[n+1][target+1];

// Initialize dp table

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

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

if (j == 0) {

dp[i][j] = true; // Sum of 0 is always possible (with empty subset)

} else if (i == 0) {

dp[i][j] = false; // No sum can be formed with zero elements

} else {

// If the current element is greater than the sum, exclude it

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

dp[i][j] = dp[i-1][j] || dp[i-1][j - set[i-1]]; // Include or exclude the element

} else {

dp[i][j] = dp[i-1][j]; // Exclude the element

}

}

}

}

return dp[n][target]; // Return the value in the bottom-right cell

}

int main() {

// Input set and target sum

int set[MAX\_SIZE] = {5, 10, 12, 13, 15, 18};

int target = TARGET;

// Number of elements in the set

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

// Solve the subset sum problem

if (isSubsetSum(set, n, target)) {

printf("Yes, there exists a subset with sum %d.\n", target);

} else {

printf("No, there is no subset with sum %d.\n", target);

}

return 0;

}

9. write a c program to solve the follwing 0/1 knapsack using dynamic programing aproach p={15,25,13,23},weight w=(2,6,12,9),knapsack c=20 and the number of item n=4

Title: write a c program to solve the follwing 0/1 knapsack using dynamic programing aproach p={15,25,13,23},weight w=(2,6,12,9),knapsack c=20 and the number of item n=4

Theory: The 0/1 Knapsack problem is a well-known combinatorial optimization problem. Given a set of items, each with a weight and a value, you need to determine the maximum value that can be accommodated in a knapsack of fixed capacity, such that the total weight does not exceed the capacity of the knapsack.

The goal is to maximize the value, given:

A set of item values:

𝑝

={15,25,13,23}

p={15,25,13,23}

Corresponding weights:

𝑤

={2,6,12,9}

w={2,6,12,9}

Knapsack capacity:

𝑐

=

20

c=20

Number of items:

𝑛

=

4

n=4

Algorithm: Create a 2D array dp[][] where dp[i][w] stores the maximum value that can be obtained by considering the first i items with a knapsack capacity w.

Initialize:

dp[0][w] = 0 for all w, because with 0 items, the maximum value is 0.

dp[i][0] = 0 for all i, because with 0 capacity, the maximum value is also 0.

For each item, check if it can be included in the knapsack (i.e., if its weight is less than or equal to the current capacity). If including the item results in a higher value, update the table.

Source code: #include <stdio.h>

#define MAX\_ITEMS 4

#define MAX\_CAPACITY 20

// Function to solve the 0/1 Knapsack problem using dynamic programming

int knapsack(int p[], int w[], int n, int c) {

int dp[n + 1][c + 1];

// Initialize the dp table

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

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

if (i == 0 || j == 0) {

dp[i][j] = 0; // Base case: no items or no capacity

} else if (w[i - 1] <= j) {

// Item i can be included or excluded

dp[i][j] = (p[i - 1] + dp[i - 1][j - w[i - 1]] > dp[i - 1][j])

? p[i - 1] + dp[i - 1][j - w[i - 1]]

: dp[i - 1][j];

} else {

// Item i cannot be included

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

}

}

}

// The bottom-right cell of the table holds the maximum value

return dp[n][c];

}

int main() {

// Item values

int p[MAX\_ITEMS] = {15, 25, 13, 23};

// Item weights

int w[MAX\_ITEMS] = {2, 6, 12, 9};

// Knapsack capacity

int c = 20;

// Number of items

int n = 4;

// Solve the knapsack problem and print the result

int maxValue = knapsack(p, w, n, c);

printf("Maximum value that can be obtained: %d\n", maxValue);

return 0;

}

10. write a c program to solve the tower of hanoi problem for the N disk

Title: write a c program to solve the tower of hanoi problem for the N disk

Theory: The Tower of Hanoi is a classic problem in recursive algorithms. It involves three rods and a number of disks of different sizes that can slide onto any rod. The objective is to move all the disks from one rod to another, with the following rules:

Only one disk can be moved at a time.

Each move consists of taking the top disk from one of the stacks and placing it on top of another stack.

No disk may be placed on top of a smaller disk

Algorithm: 1.Move N-1 disks from the source rod to the auxiliary rod, using the destination rod as an auxiliary.

2.Move the largest disk (the Nth disk) directly from the source rod to the destination rod.

3.Move the N-1 disks from the auxiliary rod to the destination rod, using the source rod as an auxiliary

Source code:. #include <stdio.h>

// Function to solve the Tower of Hanoi problem recursively

void TowerOfHanoi(int N, char source, char destination, char auxiliary) {

// Base case: If there is only one disk, move it from source to destination

if (N == 1) {

printf("Move disk 1 from %c to %c\n", source, destination);

return;

}

// Recursive case: Move N-1 disks from source to auxiliary

TowerOfHanoi(N - 1, source, auxiliary, destination);

// Move the Nth disk from source to destination

printf("Move disk %d from %c to %c\n", N, source, destination);

// Move the N-1 disks from auxiliary to destination

TowerOfHanoi(N - 1, auxiliary, destination, source);

}

int main() {

int N;

// Ask the user for the number of disks

printf("Enter the number of disks: ");

scanf("%d", &N);

// Call the Tower of Hanoi function to solve the problem

printf("\nSolution for %d disks:\n", N);

TowerOfHanoi(N, 'A', 'C', 'B'); // A is the source rod, C is the destination, B is the auxiliary rod

return 0;

}