**Department of Computer Science &**

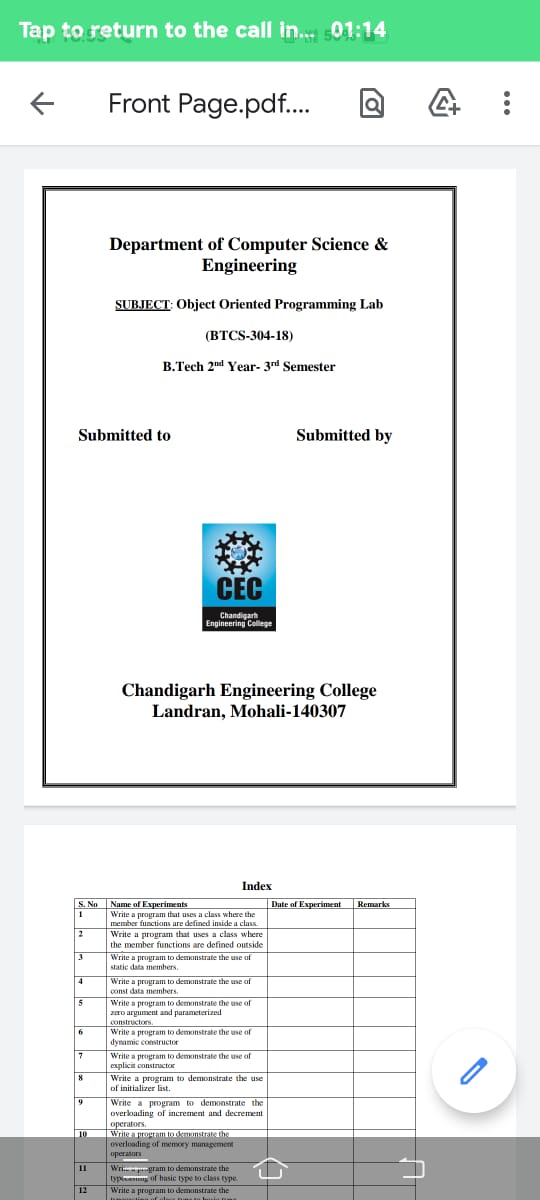
**Engineering**

Subject: Design And Analysis Of Algorithm

(BTCS-403-18)

B.Tech 2nd Year- 4th Semester

**Submitted by Submitted to**



**Chandigarh Engineering College**

**Landran, Mohali-140307**

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| S NO | EXPERIMENT NAME | DATE | SIGN |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

PRACTICAL NO. 1

Aim: Program to perform linear search.

Program:

#include <iostream>

using namespace std;

int linearsearch(int arr[], int n, int key)

{

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

{

if (arr[i] == key)

{

return i;

}

}

return -1;

}

int main()

{

int n;

cout << "Enter number of elements: ";

cin >> n;

int arr[n];

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

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

cin >> arr[i];

int key;

cout << "Enter Key: ";

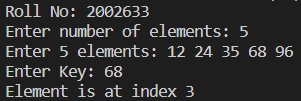
cin >> key;

cout << "Element is at index " << linearsearch(arr, n, key) << endl;

return 0;

}

Output:



PRACTICAL NO. 2

Aim: Program to perform binary search.

Program:

#include <iostream>

using namespace std;

int binarysearch(int arr[], int n, int key)

{

int s = 0;

int e = n;

while (s <= e)

{

int mid = (s + e) / 2;

if (arr[mid] == key)

return mid;

else if (arr[mid] > key)

e = mid - 1;

else

s = mid + 1;

}

return -1;

}

int main()

{

int n;

cout << "Roll No: 2002633\n";

cout << "Enter number of elements: ";

cin >> n;

int arr[n];

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

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

{

cin >> arr[i];

}

int key;

cout << "Enter Key: ";

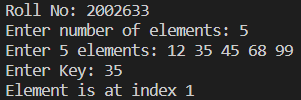
cin >> key;

cout << "Element is at index " << binarysearch(arr, n, key) << endl;

return 0;

}

Output:



PRACTICAL NO. 3

Aim: Program to perform merge sort.

Program:

#include <iostream>

using namespace std;

// Merges two subarrays of array[].

// First subarray is arr[begin..mid]

// Second subarray is arr[mid+1..end]

void merge(int array[], int const left, int const mid, int const right)

{

auto const subArrayOne = mid - left + 1;

auto const subArrayTwo = right - mid;

// Create temp arrays

auto \*leftArray = new int[subArrayOne],

\*rightArray = new int[subArrayTwo];

// Copy data to temp arrays leftArray[] and rightArray[]

for (auto i = 0; i < subArrayOne; i++)

leftArray[i] = array[left + i];

for (auto j = 0; j < subArrayTwo; j++)

rightArray[j] = array[mid + 1 + j];

auto indexOfSubArrayOne = 0, // Initial index of first sub-array

indexOfSubArrayTwo = 0; // Initial index of second sub-array

int indexOfMergedArray = left; // Initial index of merged array

// Merge the temp arrays back into array[left..right]

while (indexOfSubArrayOne < subArrayOne && indexOfSubArrayTwo < subArrayTwo)

{

if (leftArray[indexOfSubArrayOne] <= rightArray[indexOfSubArrayTwo])

{

array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];

indexOfSubArrayOne++;

}

else

{

array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];

indexOfSubArrayTwo++;

}

indexOfMergedArray++;

}

// Copy the remaining elements of

// left[], if there are any

while (indexOfSubArrayOne < subArrayOne)

{

array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];

indexOfSubArrayOne++;

indexOfMergedArray++;

}

// Copy the remaining elements of

// right[], if there are any

while (indexOfSubArrayTwo < subArrayTwo)

{

array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];

indexOfSubArrayTwo++;

indexOfMergedArray++;

}

}

// begin is for left index and end is

// right index of the sub-array

// of arr to be sorted \*/

void mergeSort(int array[], int const begin, int const end)

{

if (begin >= end)

return; // Returns recursively

auto mid = begin + (end - begin) / 2;

mergeSort(array, begin, mid);

mergeSort(array, mid + 1, end);

merge(array, begin, mid, end);

}

// UTILITY FUNCTIONS

// Function to print an array

void printArray(int A[], int size)

{

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

cout << A[i] << " ";

}

// Driver code

int main()

{

cout << "Roll No: 2002633\n";

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

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

cout << "Given array is \n";

printArray(arr, arr\_size);

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

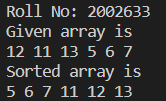
cout << "\nSorted array is \n";

printArray(arr, arr\_size);

return 0;

}

Output:



PRACTICAL NO. 4

Aim: Program to perform bubble sort.

Program:

#include <iostream>

using namespace std;

void Bubble(int arr[], int n)

{

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

{

bool swapped = false;

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

{

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

{

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

swapped = true;

}

}

if (swapped == false)

{

break;

}

}

}

int main()

{

cout << "Roll No: 2002633\n";

int arr[] = {7, 9, 1, 6, 2};

cout << "Unsorted array is: ";

for (int i = 0; i < 5; i++)

{

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

}

cout << endl;

Bubble(arr, 5);

cout << "Sorted array is: ";

for (int i = 0; i < 5; i++)

{

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

}

return 0;

}

Output:



PRACTICAL NO. 5

Aim: Program to perform 0/1 knapsack using Greedy Method.

Program:

/\* A Naive recursive implementation of

0-1 Knapsack problem \*/

#include <bits/stdc++.h>

using namespace std;

// A utility function that returns

// maximum of two integers

int max(int a, int b) { return (a > b) ? a : b; }

// Returns the maximum value that

// can be put in a knapsack of capacity W

int knapSack(int W, int wt[], int val[], int n)

{

// Base Case

if (n == 0 || W == 0)

return 0;

// If weight of the nth item is more

// than Knapsack capacity W, then

// this item cannot be included

// in the optimal solution

if (wt[n - 1] > W)

return knapSack(W, wt, val, n - 1);

// Return the maximum of two cases:

// (1) nth item included

// (2) not included

else

return max(

val[n - 1] + knapSack(W - wt[n - 1],

wt, val, n - 1),

knapSack(W, wt, val, n - 1));

}

// Driver code

int main()

{

cout << "Roll No: 2002633\n";

int val[] = {60, 100, 120};

int wt[] = {10, 20, 30};

int W = 50;

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

cout << knapSack(W, wt, val, n);

return 0;

}

Output:



PRACTICAL NO. 6

Aim: Program to perform knapsack problem using Dynamic Programming.

Program:

// A dynamic programming based

// solution for 0-1 Knapsack problem

#include <bits/stdc++.h>

using namespace std;

// A utility function that returns

// maximum of two integers

int max(int a, int b)

{

return (a > b) ? a : b;

}

// Returns the maximum value that

// can be put in a knapsack of capacity W

int knapSack(int W, int wt[], int val[], int n)

{

int i, w;

vector<vector<int>> K(n + 1, vector<int>(W + 1));

// Build table K[][] in bottom up manner

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

{

for (w = 0; w <= W; w++)

{

if (i == 0 || w == 0)

K[i][w] = 0;

else if (wt[i - 1] <= w)

K[i][w] = max(val[i - 1] +

K[i - 1][w - wt[i - 1]],

K[i - 1][w]);

else

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

}

}

return K[n][W];

}

// Driver Code

int main()

{

cout << "Roll No: 2002633\n";

int val[] = {60, 100, 120};

int wt[] = {10, 20, 30};

int W = 50;

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

cout << knapSack(W, wt, val, n);

return 0;

}

Output:



PRACTICAL NO. 7

Aim: Program to perform Matrix Chain Multiplication Using Dynamic Programming.

Program:

#include <bits/stdc++.h>

using namespace std;

int dp[100][100];

int matrixChainMemoised(int \*p, int i, int j)

{

if (i == j)

{

return 0;

}

if (dp[i][j] != -1)

{

return dp[i][j];

}

dp[i][j] = INT\_MAX;

for (int k = i; k < j; k++)

{

dp[i][j] = min(

dp[i][j], matrixChainMemoised(p, i, k) + matrixChainMemoised(p, k + 1, j) + p[i - 1] \* p[k] \* p[j]);

}

return dp[i][j];

}

int MatrixChainOrder(int \*p, int n)

{

int i = 1, j = n - 1;

return matrixChainMemoised(p, i, j);

}

int main()

{

cout << "Roll No: 2002633\n";

int arr[] = {1, 2, 3, 4};

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

memset(dp, -1, sizeof dp);

cout << "Minimum number of multiplications is "

<< MatrixChainOrder(arr, n);

}

Output:



PRACTICAL NO. 8

Aim: Program to code and analyze to find an optimal solution to TSP using Dynamic Programming.

Program:

#include <bits/stdc++.h>

using namespace std;

#define V 4

// implementation of traveling Salesman Problem

int travllingSalesmanProblem(int graph[][V], int s)

{

// store all vertex apart from source vertex

vector<int> vertex;

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

if (i != s)

vertex.push\_back(i);

// store minimum weight Hamiltonian Cycle.

int min\_path = INT\_MAX;

do

{

// store current Path weight(cost)

int current\_pathweight = 0;

// compute current path weight

int k = s;

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

{

current\_pathweight += graph[k][vertex[i]];

k = vertex[i];

}

current\_pathweight += graph[k][s];

// update minimum

min\_path = min(min\_path, current\_pathweight);

} while (

next\_permutation(vertex.begin(), vertex.end()));

return min\_path;

}

// Driver Code

int main()

{

cout << "Roll No: 2002633\n";

// matrix representation of graph

int graph[][V] = {{0, 10, 15, 20},

{10, 0, 35, 25},

{15, 35, 0, 30},

{20, 25, 30, 0}};

int s = 0;

cout << travllingSalesmanProblem(graph, s) << endl;

return 0;

}

Output:



PRACTICAL NO. 9

Aim: Program to implement an application of DFS.

Program: