DAA Lab file

### A PRACTICAL FILE SUBMITTED IN PARTIAL FULFILLMENT OF COURSE

Design and Analysis of Algorithm Lab

**BACHELORS OF TECHNOLOGY**

## Department of Information Technology

**SUBMITTED BY**

Nitin sharma 1905372

**SUBMITTED TO**

## Dr. Manjot Kaur Session - Jan-June 2022

## GURU NANAK DEV ENGINEERING COLLEGE LUDHIANA-141006, INDIA

**Index Page**

# Implement binary search algorithm and compute its time complexity.



#include <bits/stdc++.h> using namespace std;

int binarySearch(int arr[], int l, int r, int x)

{

if (r >= l) {

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

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

if (arr[mid] > x)

return binarySearch(arr, l, mid - 1, x);

return binarySearch(arr, mid + 1, r, x);

}

return -1;

}

int main(void)

{

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

int x = 10;

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

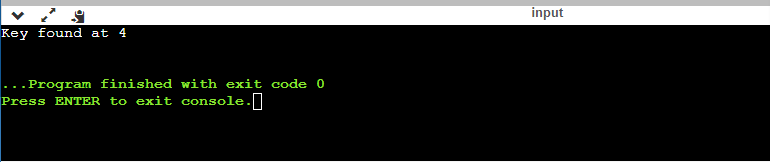
int result = binarySearch(arr, 0, n - 1, x); (result == -1)

? cout << "Element is not present in array"

: cout << "Element is present at index " << result; return 0;

}

#### Output:-



#### Time Complexity:-

##### At every step, array is divided into half. So total log n comparisons will do done for n

##### elements in worst case (when element is found at last partition). In best case, only one comparison will be done as the element found at mid of array.

# Implement merge sort algorithm and demonstrate divide and conquer technique.



class MergeSort

{

// Merges two subarrays of arr[].

// First subarray is arr[l..m]

// Second subarray is arr[m+1..r]

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

{

// Find sizes of two subarrays to be merged int n1 = m - l + 1;

int n2 = r - m;

/\* Create temp arrays \*/ int L[] = new int[n1]; int R[] = new int[n2];

/\*Copy data to temp arrays\*/ 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];

/\* Merge the temp arrays \*/

// Initial indexes of first and second subarrays int i = 0, j = 0;

// Initial index of merged subarray array int k = l;

while (i < n1 && j < n2) { if (L[i] <= R[j]) {

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

}

else {

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

} k++;

}

/\* Copy remaining elements of L[] if any \*/ while (i < n1) {

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

k++;

}

/\* Copy remaining elements of R[] if any \*/ while (j < n2) {

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

k++;

}

}

// Main function that sorts arr[l..r] using

// merge()

void sort(int arr[], int l, int r)

{

if (l < r) {

// Find the middle point int m =l+ (r-l)/2;

// Sort first and second halves sort(arr, l, m);

sort(arr, m + 1, r);

// Merge the sorted halves merge(arr, l, m, r);

}

}

public static void main(String args[])

{

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

System.out.println("Given Array"); printArray(arr);

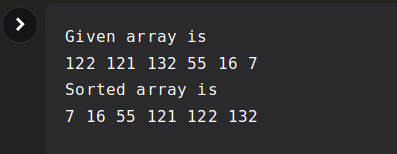
MergeSort ob = new MergeSort(); ob.sort(arr, 0, arr.length - 1);

System.out.println("\nSorted array"); printArray(arr);

}

}

#### Output:-



#### Time Complexity:-

##### Worst Case Complexity :- O(n log n)

# Analyse the complexity of Quick sort algorithm



#include <bits/stdc++.h> using namespace std;

void swap(int\* a, int\* b)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

int partition (int arr[], int low, int high)

{

int pivot = arr[high]; // pivot int i = (low - 1);

for (int j = low; j <= high - 1; j++)

{

// If current element is smaller than the pivot if (arr[j] < pivot)

{

i++; // increment index of smaller element swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]); return (i + 1);

}

void quickSort(int arr[], int low, int high)

{

if (low < high)

{

/\* pi is partitioning index, arr[p] is now at right place \*/

int pi = partition(arr, low, high);

// Separately sort elements before

// partition and after partition quickSort(arr, low, pi - 1); quickSort(arr, pi + 1, high);

}

}

/\* Function to print an array \*/ void printArray(int arr[], int size)

{

int i;

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

cout << endl;

}

int main()

{

int arr[] = {10, 7, 8, 9, 1, 5};

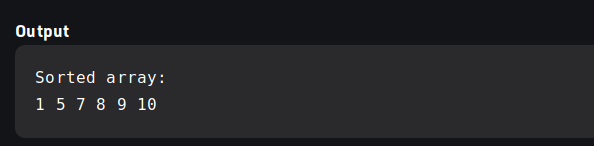
int n = sizeof(arr) / sizeof(arr[0]); quickSort(arr, 0, n - 1);

cout << "Sorted array: \n"; printArray(arr, n);

return 0;

}

#### Output:-



#### Time Complexity:-

##### Worst Case Complexity :- O(*n*2)

##### Worst Case Complexity :- O(*nlogn*)

# Implement Minimum cost spanning tree using Greedy Algorithm



#include <bits/stdc++.h> using namespace std;

// DSU data structure

// path compression + rank by union

class DSU { int\* parent; int\* rank;

public:

DSU(int n)

{

parent = new int[n]; rank = new int[n];

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

rank[i] = 1;

}

}

// Find function int find(int i)

{

if (parent[i] == -1) return i;

return parent[i] = find(parent[i]);

}

// union function

void unite(int x, int y)

{

int s1 = find(x); int s2 = find(y);

if (s1 != s2) {

if (rank[s1] < rank[s2]) { parent[s1] = s2; rank[s2] += rank[s1];

}

else {

parent[s2] = s1; rank[s1] += rank[s2];

}

}

}

};

class Graph {

vector<vector<int> > edgelist; int V;

public:

Graph(int V) { this->V = V; }

void addEdge(int x, int y, int w)

{

edgelist.push\_back({ w, x, y });

}

void kruskals\_mst()

{

// 1. Sort all edges sort(edgelist.begin(), edgelist.end());

// Initialize the DSU DSU s(V);

int ans = 0;

cout << "Following are the edges in the " "constructed MST"

<< endl;

for (auto edge : edgelist) { int w = edge[0];

int x = edge[1]; int y = edge[2];

// take that edge in MST if it does form a cycle if (s.find(x) != s.find(y)) {

s.unite(x, y); ans += w;

cout << x << " -- " << y << " == " << w

<< endl;

}

}

cout << "Minimum Cost Spanning Tree: " << ans;

}

};

int main()

{

/\* Let us create following weighted graph

10

0--------1

| \ | 6| 5\ |15

| \ |

2--------3

4 \*/ Graph g(4);

|  |  |  |
| --- | --- | --- |
| g.addEdge(0, | 1, | 10); |
| g.addEdge(1, | 3, | 15); |
| g.addEdge(2, | 3, | 4); |
| g.addEdge(2, | 0, | 6); |
| g.addEdge(0, | 3, | 5); |

// int n, m;

// cin >> n >> m;

// Graph g(n);

// for (int i = 0; i < m; i++)

// {

// int x, y, w;

// cin >> x >> y >> w;

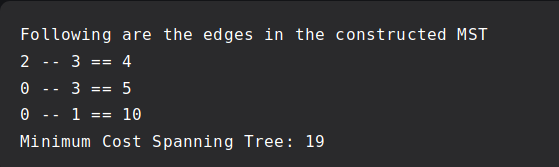
// g.addEdge(x, y, w);

// }

g.kruskals\_mst(); return 0;

}

#### Output:-



#### Time Complexity:-

##### Worst Case Complexity :- O(e log e)

# Practical - 5

# Implement greedy algorithm to solve single-source shortest path problem

// A C++ program for Dijkstra's single source shortest path algorithm.

// The program is for adjacency matrix representation of the graph

#include <iostream>

using namespace std;

#include <limits.h>

// Number of vertices in the graph

#define V 9

// A utility function to find the vertex with minimum distance value, from

// the set of vertices not yet included in shortest path tree

int minDistance(int dist[], bool sptSet[])

{

// Initialize min value

int min = INT\_MAX, min\_index;

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

if (sptSet[v] == false && dist[v] <= min)

min = dist[v], min\_index = v;

return min\_index;

}

// A utility function to print the constructed distance array

void printSolution(int dist[])

{

cout <<"Vertex \t Distance from Source" << endl;

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

cout << i << " \t\t"<<dist[i]<< endl;

}

// Function that implements Dijkstra's single source shortest path algorithm

// for a graph represented using adjacency matrix representation

void dijkstra(int graph[V][V], int src)

{

int dist[V]; // The output array. dist[i] will hold the shortest

// distance from src to i

bool sptSet[V]; // sptSet[i] will be true if vertex i is included in shortest

// path tree or shortest distance from src to i is finalized

// Initialize all distances as INFINITE and stpSet[] as false

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

dist[i] = INT\_MAX, sptSet[i] = false;

// Distance of source vertex from itself is always 0

dist[src] = 0;

// Find shortest path for all vertices

for (int count = 0; count < V - 1; count++) {

// Pick the minimum distance vertex from the set of vertices not

// yet processed. u is always equal to src in the first iteration.

int u = minDistance(dist, sptSet);

// Mark the picked vertex as processed

sptSet[u] = true;

// Update dist value of the adjacent vertices of the picked vertex.

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

// Update dist[v] only if is not in sptSet, there is an edge from

// u to v, and total weight of path from src to v through u is

// smaller than current value of dist[v]

if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX

&& dist[u] + graph[u][v] < dist[v])

dist[v] = dist[u] + graph[u][v];

}

// print the constructed distance array

printSolution(dist);

}

// driver program to test above function

int main()

{

/\* Let us create the example graph discussed above \*/

int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },

{ 4, 0, 8, 0, 0, 0, 0, 11, 0 },

{ 0, 8, 0, 7, 0, 4, 0, 0, 2 },

{ 0, 0, 7, 0, 9, 14, 0, 0, 0 },

{ 0, 0, 0, 9, 0, 10, 0, 0, 0 },

{ 0, 0, 4, 14, 10, 0, 2, 0, 0 },

{ 0, 0, 0, 0, 0, 2, 0, 1, 6 },

{ 8, 11, 0, 0, 0, 0, 1, 0, 7 },

{ 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

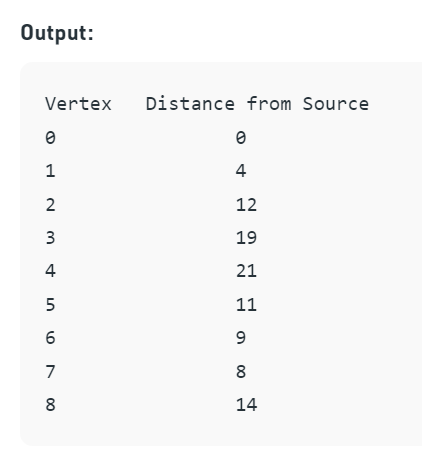
dijkstra(graph, 0);

return 0;

}

// This code is contributed by shivanisinghss2110

Output:



**PRACTICAL - 6**

**Use dynamic programming to solve 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()

{

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 -7**

**Solve all pairs shortest path problem using dynamic programming.**

// C++ Program for Floyd Warshall Algorithm

#include <bits/stdc++.h>

using namespace std;

// Number of vertices in the graph

#define V 4

/\* Define Infinite as a large enough

value.This value will be used for

vertices not connected to each other \*/

#define INF 99999

// A function to print the solution matrix

void printSolution(int dist[][V]);

// Solves the all-pairs shortest path

// problem using Floyd Warshall algorithm

void floydWarshall(int graph[][V])

{

/\* dist[][] will be the output matrix

that will finally have the shortest

distances between every pair of vertices \*/

int dist[V][V], i, j, k;

/\* Initialize the solution matrix same

as input graph matrix. Or we can say

the initial values of shortest distances

are based on shortest paths considering

no intermediate vertex. \*/

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

for (j = 0; j < V; j++)

dist[i][j] = graph[i][j];

/\* Add all vertices one by one to

the set of intermediate vertices.

---> Before start of an iteration,

we have shortest distances between all

pairs of vertices such that the

shortest distances consider only the

vertices in set {0, 1, 2, .. k-1} as

intermediate vertices.

----> After the end of an iteration,

vertex no. k is added to the set of

intermediate vertices and the set becomes {0, 1, 2, ..

k} \*/

for (k = 0; k < V; k++) {

// Pick all vertices as source one by one

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

// Pick all vertices as destination for the

// above picked source

for (j = 0; j < V; j++) {

// If vertex k is on the shortest path from

// i to j, then update the value of

// dist[i][j]

if (dist[i][j] > (dist[i][k] + dist[k][j])

&& (dist[k][j] != INF

&& dist[i][k] != INF))

dist[i][j] = dist[i][k] + dist[k][j];

}

}

}

// Print the shortest distance matrix

printSolution(dist);

}

/\* A utility function to print solution \*/

void printSolution(int dist[][V])

{

cout << "The following matrix shows the shortest "

"distances"

" between every pair of vertices \n";

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

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

if (dist[i][j] == INF)

cout << "INF"

<< " ";

else

cout << dist[i][j] << " ";

}

cout << endl;

}

}

// Driver code

int main()

{

/\* Let us create the following weighted graph

10

(0)------->(3)

| /|\

5 | |

| | 1

\|/ |

(1)------->(2)

3 \*/

int graph[V][V] = { { 0, 5, INF, 10 },

{ INF, 0, 3, INF },

{ INF, INF, 0, 1 },

{ INF, INF, INF, 0 } };

// Print the solution

floydWarshall(graph);

return 0;

}

Output:

