

MAHARAJA SURAJMAL INSTITUTE OF TECHNOLOGY

ALGORITHM DESIGN AND ANALYSIS LAB (ETCS-351)

SUBMITTED TO:

**Mrs. Jyoti Khurana**

SUBMITTED BY:

## Name: Aakriti Sharma

## Class: IT-2

## S No. : 47

Enrollment No: 20715003119

**INDEX**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.NO** | **NAME OF EXPERIMENT** | **PAGE NO.** | **DATE OF EXPERIMENT** | **REMARKS** |
| 1) | To sort an array using Insertion Sort and analyse its time complexity. | 3-6 | 25/08/2021 |  |
| 2) | To sort an array using Merge Sort and analyse its time complexity. | 7-11 | 01/09/2021 |  |
| 3) | To sort an array using Quick Sort and analyse its time complexity. | 12-14 | 08/09/2021 |  |
| 4) | To implement Linear Search and analyse its time complexity. | 15-17 | 15/09/2021 |  |
| 5) | To implement Binary Search and analyse its time complexity. | 18-20 | 15/09/2021 |  |
| 6) | To implement Matrix Multiplication and analyse its time complexity. | 21-23 | 22-09-2021 |  |
| 7) | To implement Longest Common Subsequence and analyse its time complexity. | 24-26 | 22-09-2021 |  |
| 8) | To implement Huffman Coding and analyse its complexity. |  |  |  |
| 9) | To implement Optimal Binary Search tree and analyse its complexity. |  |  |  |
| 10) | To implement Prim’s algorithm and analyse its complexity. |  |  |  |
| 11) | To implement Dijkstra’s Algorithm and analyse its complexity. |  |  |  |
| 12) | N |  |  |  |

EXPERIMENT-1

AIM : - To sort an array using Insertion Sort and analyse its time complexity.

Code:-

#include <iostream>

#include <ctime>

using namespace std;

int main()

{

int arr[50], tot, i, j, k, elem, index;

clock\_t startTime,endTime;

cout<<"Enter the Size for Array: ";

cin>>tot;

cout<<"Enter "<<tot<<" Array Elements: ";

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

cin>>arr[i];

startTime = clock();

for(i=1; i<tot; i++)

{

elem = arr[i];

if(elem<arr[i-1])

{

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

{

if(elem<arr[j])

{

index = j;

for(k=i; k>j; k--)

arr[k] = arr[k-1];

break;

}

}

}

else

continue;

arr[index] = elem;

}

endTime = clock()-startTime;

cout<<"\nThe New Array (Sorted Array):\n";

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

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

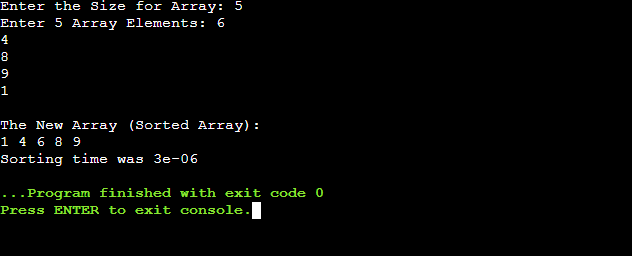
cout<<endl;

cout<<"Sorting time was "<<(float)endTime/CLOCKS\_PER\_SEC;

return 0;

}

OUTPUT

****

EXPERIMENT-2

AIM:- To sort an array using Merge Sort and analyse its time complexity.

Code:-

#include <bits/stdc++.h>

using namespace std;

void Merge(int \*a, int low, int high, int mid)

{

int i, j, k, temp[high-low+1];

i = low;

k = 0;

j = mid + 1;

while (i <= mid && j <= high)

{

if (a[i] < a[j])

{

temp[k] = a[i];

k++;

i++;

}

else

{

temp[k] = a[j];

k++;

j++;

}

}

while (i <= mid)

{

temp[k] = a[i];

k++;

i++;

}

while (j <= high)

{

temp[k] = a[j];

k++;

j++;

}

for (i = low; i <= high; i++)

{

a[i] = temp[i-low];

}

}

void MergeSort(int \*a, int low, int high)

{

int mid;

if (low < high)

{

mid=(low+high)/2;

MergeSort(a, low, mid);

MergeSort(a, mid+1, high);

Merge(a, low, high, mid);

}

}

int main()

{

int n, i;

clock\_t start, end;

cout<<"\nEnter the number of elements to be sorted: ";

cin>>n;

int arr[n];

cout<<"Enter "<<n<<" element: ";

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

{

cin>>arr[i];

}

start = clock();

MergeSort(arr, 0, n-1);

cout<<"\nSorted Data ";

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

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

end = clock();

double time\_taken = double(end - start) / double(CLOCKS\_PER\_SEC);

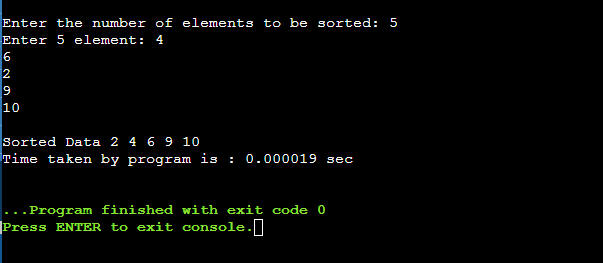
cout << "\nTime taken by program is : " << fixed << time\_taken << setprecision(5);

cout << " sec " << endl;

return 0;

}

OUTPUT



EXPERIMENT-3

AIM:- To sort an array using Quick Sort and analyse its time complexity.

Code:-

#include <iostream>

#include <ctime>

using namespace std;

int partition(int \*a,int start,int end)

{

int pivot=a[end];

int P\_index=start;

int i,t;

for(i=start;i<end;i++)

{

if(a[i]<=pivot)

{

t=a[i];

a[i]=a[P\_index];

a[P\_index]=t;

P\_index++;

}

}

t=a[end];

a[end]=a[P\_index];

a[P\_index]=t;

return P\_index;

}

void Quicksort(int \*a,int start,int end)

{

if(start<end)

{

int P\_index=partition(a,start,end);

Quicksort(a,start,P\_index-1);

Quicksort(a,P\_index+1,end);

}

}

int main()

{

clock\_t startTime,endTime;

startTime = clock();

int n;

cout<<"Enter number of elements: ";

cin>>n;

int a[n];

cout<<"Enter the array elements:\n";

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

{

cin>>a[i];

}

startTime = clock();

Quicksort(a,0,n-1);

cout<<"After Quick Sort the array is:\n";

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

{

cout<<a[i]<<" ";

}

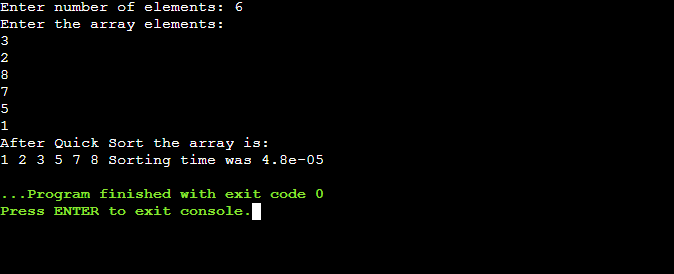
endTime = clock()-startTime;

cout<<"Sorting time was "<<(float)endTime/CLOCKS\_PER\_SEC;

return 0;

}

OUTPUT



EXPERIMENT-4

AIM : - To implement Linear Search and analyse its time complexity.

#include<iostream>

#include<time.h>

using namespace std;

int main(){

clock\_t start,end;

const double CLK\_TCK = 1000.0;

int n, item;

int flag = -1; cout<<"Enter n:"<<endl; cin>>n;

int arr[n] = {0};

cout<<"Enter elements:"<<endl;

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

{

cin>>arr[i];

}

cout<<"Enter item to search: "<<endl;

cin>>item;

start = clock();

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

if(arr[i] == item)

{

flag = i;

}

end = clock();

}

if(flag == -1){

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

} else

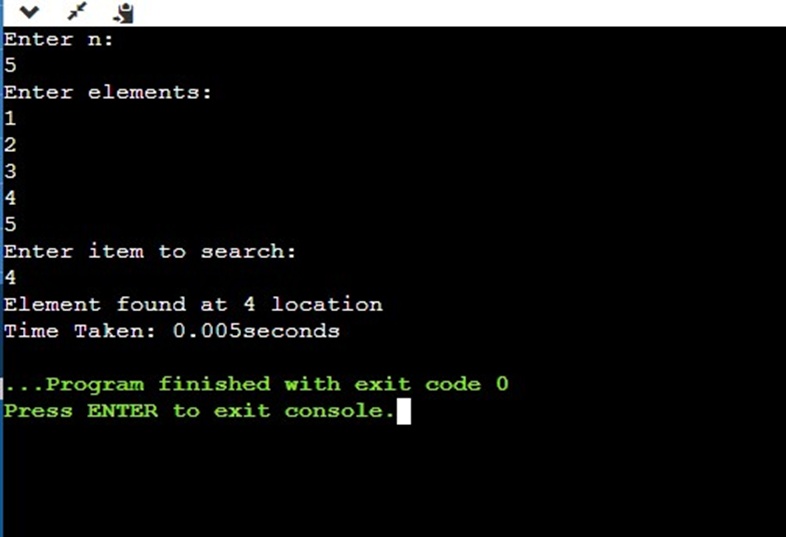
{

cout<<"Element found at "<<(flag+1)<<" location"<<endl;

}

cout<<"Time Taken: "<<(end-start)/CLK\_TCK<<"seconds";

}



EXPERIMENT-5

AIM : - To implement Binary Search and analyse its time complexity.

#include <iostream>

#include<time.h>

using namespace std;

int binary(int arr[], int x, int y, int temp)

{

while(x <= y)

int m = x + (y - x)/2;

if (arr[m] == temp) return m;

if (arr[m] < temp) x = m+1;

else y = m - 1;

}

return -1;

}

int main()

{

clock\_t start,end; const double CLK\_TCK = 1000.0;

int n;

cout<<"Enter n:"<<endl;

cin>>n;

int arr[n] = {0};

cout<<"Enter elements of array:"<<endl;

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

cin>>arr[i];

}

int temp = 10;

cout<<"Enter item to search:"<<endl;

cin>>temp;

start = clock();

int item = binary(arr, 0, n - 1, temp);

end = clock();

if(item == -1)

{

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

else

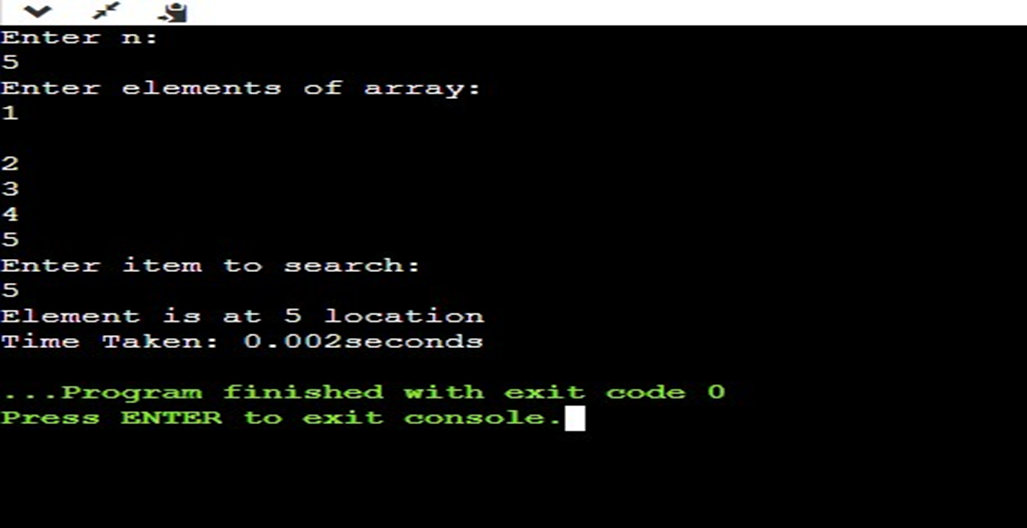
{ cout<<"Element is at "<<(item+1)<<" location "<<endl;

}

cout<<"Time Taken: "<<(end-start)/CLK\_TCK<<"seconds";

return 0;

}



EXPERIMENT-6

AIM : - To implement Matrix Multiplication and analyse its time complexity.

#include<bits/stdc++.h>

#include<time.h>

using namespace std;

int mcm(int p[], int n)

{

int m[n][n];

int i, j, k, l, q;

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

m[i][i] = 0;

for (l=2; l<n; l++)

{

for (i=1; i<n-l+1; i++)

{ j = i+l-1;

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

for (k=i; k<=j-1; k++)

{

q = m[i][k] + m[k+1][j] + p[i-1]\*p[k]\*p[j];

if (q < m[i][j])

m[i][j] = q;

}

}

}

return m[1][n-1];

}

int main()

{

clock\_t start,end; const double CLK\_TCK = 1000.0;

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

int size = 5;

start = clock();

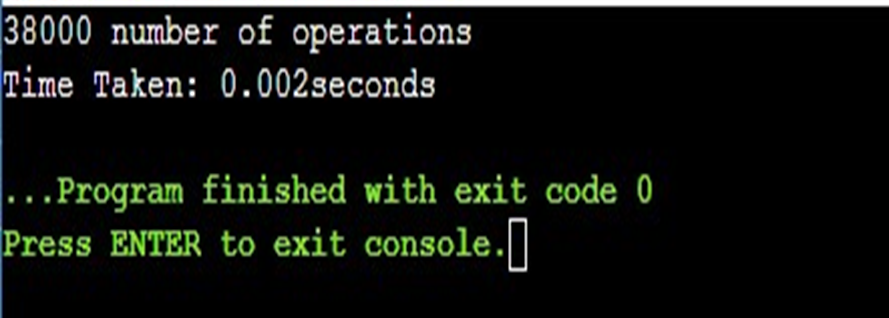
long x = mcm(arr,size);

end = clock();

cout<<x<<" number of operations"<<endl;

cout<<"Time Taken: "<<(end-start)/CLK\_TCK<<"seconds";

}



EXPERIMENT-7

AIM : - To implement Longest Common Subsequence and analyse its time complexity.

#include <iostream>

#include<string.h>

#include<time.h>

using namespace std;

void lcs(char \*a, char \*b, int m, int n)

{ int table[m + 1][n + 1];

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

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

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

table[i][j] = 0;

else if (a[i - 1] == b[j - 1])

table[i][j] = table[i - 1][j - 1] + 1;

else table[i][j] = max(table[i - 1][j], table[i][j - 1]);

}

}

int k = table[m][n];

char lcs[k + 1];

lcs[k] = '\0';

int i = m, j = n;

while (i > 0 && j > 0)

{ if (a[i - 1] == b[j - 1])

{ lcs[k - 1] = a[i - 1];

i--;

j--;

k--;

}

else if (table[i - 1][j] > table[i][j - 1])

i--;

else

j--;

}

cout<<lcs<<endl;

}

int main() { clock\_t start,end; const double CLK\_TCK = 1000.0; char a[] = "LONGEST";

char b[] = "STONE";

int m = strlen(a);

int n = strlen(b);

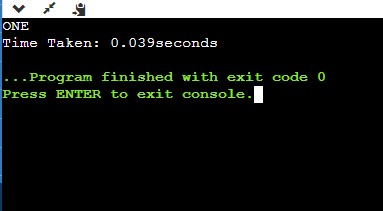
start = clock();

lcs(a, b, m, n);

end = clock();

cout<<"Time Taken: "<<(end-start)/CLK\_TCK<<"seconds";

}



EXPERIMENT-8

AIM : - To implement Huffman Coding and analyse its complexity.

#include <iostream>

#include<time.h>

using namespace std;

#define MAX\_TREE\_HT 50

struct MinHNode {

unsigned freq;

char item;

struct MinHNode \*left, \*right;

};

struct MinH {

unsigned size;

unsigned capacity;

struct MinHNode \*\*array;

};

struct MinHNode \*newNode(char item, unsigned freq) {

struct MinHNode \*temp = (struct MinHNode \*)malloc(sizeof(struct MinHNode));

temp->left = temp->right = NULL;

temp->item = item;

temp->freq = freq;

return temp;

}

struct MinH \*createMinH(unsigned capacity) {

struct MinH \*minHeap = (struct MinH \*)malloc(sizeof(struct MinH));

minHeap->size = 0;

minHeap->capacity = capacity;

minHeap->array = (struct MinHNode \*\*)malloc(minHeap->capacity \* sizeof(struct MinHNode \*));

return minHeap;

}

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

int i;

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

cout << arr[i];

cout << "\n";

}

// Swap function

void swapMinHNode(struct MinHNode \*\*a, struct MinHNode \*\*b) {

struct MinHNode \*t = \*a;

\*a = \*b;

\*b = t;

}

void minHeapify(struct MinH \*minHeap, int idx) {

int smallest = idx;

int left = 2 \* idx + 1;

int right = 2 \* idx + 2;

if (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]->freq)

smallest = left;

if (right < minHeap->size && minHeap->array[right]->freq < minHeap->array[smallest]->freq)

smallest = right;

if (smallest != idx) {

swapMinHNode(&minHeap->array[smallest],

&minHeap->array[idx]);

minHeapify(minHeap, smallest);

}

}

int checkSizeOne(struct MinH \*minHeap) {

return (minHeap->size == 1);

}

struct MinHNode \*extractMin(struct MinH \*minHeap) {

struct MinHNode \*temp = minHeap->array[0];

minHeap->array[0] = minHeap->array[minHeap->size - 1];

--minHeap->size;

minHeapify(minHeap, 0);

return temp;

}

void insertMinHeap(struct MinH \*minHeap, struct MinHNode \*minHeapNode) {

++minHeap->size;

int i = minHeap->size - 1;

while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {

minHeap->array[i] = minHeap->array[(i - 1) / 2];

i = (i - 1) / 2;

}

minHeap->array[i] = minHeapNode;

}

void buildMinHeap(struct MinH \*minHeap) {

int n = minHeap->size - 1;

int i;

for (i = (n - 1) / 2; i >= 0; --i)

minHeapify(minHeap, i);

}

int isLeaf(struct MinHNode \*root) {

return !(root->left) && !(root->right);

}

struct MinH \*createAndBuildMinHeap(char item[], int freq[], int size) {

struct MinH \*minHeap = createMinH(size);

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

minHeap->array[i] = newNode(item[i], freq[i]);

minHeap->size = size;

buildMinHeap(minHeap);

return minHeap;

}

struct MinHNode \*buildHfTree(char item[], int freq[], int size) {

struct MinHNode \*left, \*right, \*top;

struct MinH \*minHeap = createAndBuildMinHeap(item, freq, size);

while (!checkSizeOne(minHeap)) {

left = extractMin(minHeap);

right = extractMin(minHeap);

top = newNode('$', left->freq + right->freq);

top->left = left;

top->right = right;

insertMinHeap(minHeap, top);

}

return extractMin(minHeap);

}

void printHCodes(struct MinHNode \*root, int arr[], int top) {

if (root->left) {

arr[top] = 0;

printHCodes(root->left, arr, top + 1);

}

if (root->right) {

arr[top] = 1;

printHCodes(root->right, arr, top + 1);

}

if (isLeaf(root)) {

cout << root->item << " | ";

printArray(arr, top);

}

}

void HuffmanCodes(char item[], int freq[], int size) {

struct MinHNode \*root = buildHfTree(item, freq, size);

int arr[MAX\_TREE\_HT], top = 0;

printHCodes(root, arr, top);

}

int main() {

clock\_t start,end; const double CLK\_TCK = 1000.0;

char arr[] = {'A', 'B', 'C', 'D','E','F'};

int freq[] = {5, 1, 6, 3,16,45};

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

start = clock();

HuffmanCodes(arr, freq, size);

end = clock();

cout<<"Time Taken: "<<(end-start)/CLK\_TCK<<"seconds";

}



EXPERIMENT-9

AIM : - To implement Optimal Binary Search tree and analyse its complexity.

#include<bits/stdc++.h>

using namespace std;

int sum(int freq[], int i, int j);

int optimalSearchTree(int keys[], int freq[], int n)

{

int cost[n][n];

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

cost[i][i] = freq[i];

for (int L = 2; L <= n; L++)

{

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

{

int j = i+L-1;

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

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

{

int c = ((r > i)? cost[i][r-1]:0) +

((r < j)? cost[r+1][j]:0) +

sum(freq, i, j);

if (c < cost[i][j])

cost[i][j] = c;

}

}

}

return cost[0][n-1];

}

int sum(int freq[], int i, int j)

{

int s = 0;

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

s += freq[k];

return s;

}

int main()

{

clock\_t start, end;

const double CLK\_TCK = 1000.0;

int keys[] = {10, 12, 20};

int freq[] = {34, 8, 50};

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

start = clock();

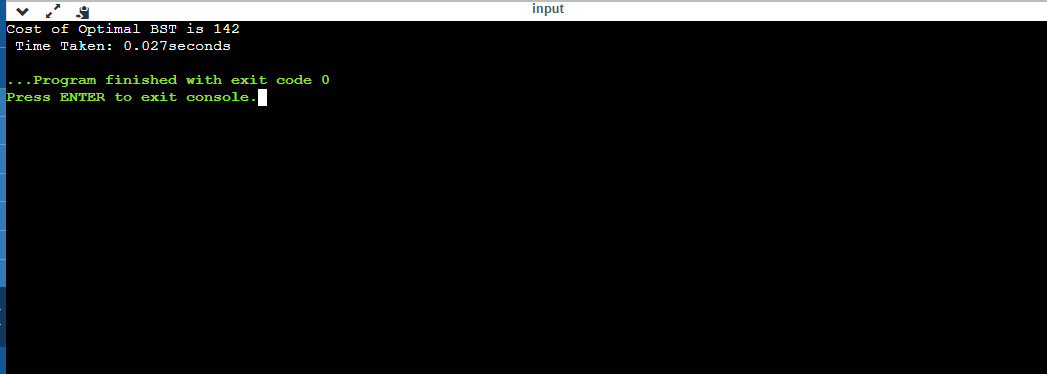
cout << "Cost of Optimal BST is " << optimalSearchTree(keys, freq, n);

end=clock();

cout<<"\n Time Taken: "<<(end-start)/CLK\_TCK<<"seconds";

return 0;

}



EXPERIMENT-10

AIM : To implement Prim’s algorithm and analyse its complexity.

#include<bits/stdc++.h>

using namespace std;

#define V 5

int minKey(int key[], bool mstSet[])

{

int min = INT\_MAX, min\_index;

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

if (mstSet[v] == false && key[v] < min)

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

return min\_index;

}

void printMST(int parent[], int graph[V][V])

{

cout<<"Edge \tWeight\n";

for (int i = 1; i < V; i++)

cout<<parent[i]<<" - "<<i<<" \t"<<graph[i][parent[i]]<<" \n";

}

void primMST(int graph[V][V])

{

int parent[V];

int key[V];

bool mstSet[V];

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

key[i] = INT\_MAX, mstSet[i] = false;

key[0] = 0;

parent[0] = -1;

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

{

int u = minKey(key, mstSet);

mstSet[u] = true;

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

if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])

parent[v] = u, key[v] = graph[u][v];

}

printMST(parent, graph);

}

int main()

{

clock\_t start, end;

const double CLK\_TCK = 1000.0;

double t;

int graph[V][V] = { { 0, 2, 0, 6, 0 },

{ 2, 0, 3, 8, 5 },

{ 0, 3, 0, 0, 7 },

{ 6, 8, 0, 0, 9 },

{ 0, 5, 7, 9, 0 } };

start=clock();

primMST(graph);

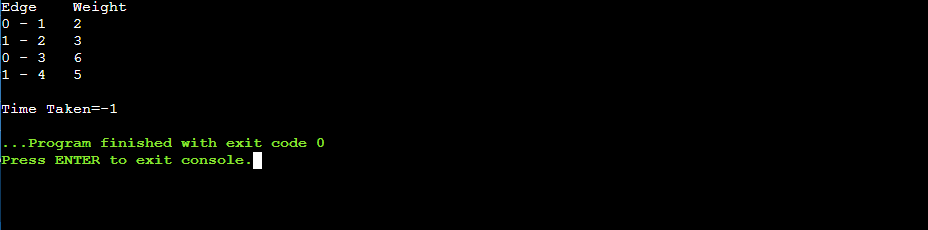
end=clock();

t= (end-start)/CLOCKS\_PER\_SEC;

cout<<endl<<"Time Taken="<<t-1;

return 0;

}



EXPERIMENT-11

AIM:- To implement Dijkstra’s Algorithm and analyse its complexity.

#include <bits/stdc++.h>

#include<dos.h>

#include<time.h>

#include<windows.h>

#include <iostream>

#include <cstdlib>

#include <vector>

#define INT\_MAX 10000000

using namespace std;

void DijkstrasTest();

int main() {

clock\_t start, end;

double t;

start=clock();

DijkstrasTest();

Sleep(1000);

end=clock();

t=((double) (end-start))/CLOCKS\_PER\_SEC;

cout<<endl<<"Time Taken="<<t-1;

return 0;

}

class Node;

class Edge;

void Dijkstras();

vector<Node\*>\* AdjacentRemainingNodes(Node\* node);

Node\* ExtractSmallest(vector<Node\*>& nodes);

int Distance(Node\* node1, Node\* node2);

bool Contains(vector<Node\*>& nodes, Node\* node);

void PrintShortestRouteTo(Node\* destination);

vector<Node\*> nodes;

vector<Edge\*> edges;

class Node {

public:

Node(char id)

: id(id), previous(NULL), distanceFromStart(INT\_MAX) {

nodes.push\_back(this);

}

public:

char id;

Node\* previous;

int distanceFromStart;

};

class Edge {

public:

Edge(Node\* node1, Node\* node2, int distance)

: node1(node1), node2(node2), distance(distance) {

edges.push\_back(this);

}

bool Connects(Node\* node1, Node\* node2) {

return (

(node1 == this->node1 &&

node2 == this->node2) ||

(node1 == this->node2 &&

node2 == this->node1));

}

public:

Node\* node1;

Node\* node2;

int distance;

};

void DijkstrasTest() {

Node\* a = new Node('a');

Node\* b = new Node('b');

Node\* c = new Node('c');

Node\* d = new Node('d');

Node\* e = new Node('e');

Node\* f = new Node('f');

Node\* g = new Node('g');

Edge\* e1 = new Edge(a, c, 1);

Edge\* e2 = new Edge(a, d, 2);

Edge\* e3 = new Edge(b, c, 2);

Edge\* e4 = new Edge(c, d, 1);

Edge\* e5 = new Edge(b, f, 3);

Edge\* e6 = new Edge(c, e, 3);

Edge\* e7 = new Edge(e, f, 2);

Edge\* e8 = new Edge(d, g, 1);

Edge\* e9 = new Edge(g, f, 1);

a->distanceFromStart = 0;

Dijkstras();

PrintShortestRouteTo(f);

}

void Dijkstras() {

while (nodes.size() > 0) {

Node\* smallest = ExtractSmallest(nodes);

vector<Node\*>\* adjacentNodes =

AdjacentRemainingNodes(smallest);

const int size = adjacentNodes->size();

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

Node\* adjacent = adjacentNodes->at(i);

int distance = Distance(smallest, adjacent) +

smallest->distanceFromStart;

if (distance < adjacent->distanceFromStart) {

adjacent->distanceFromStart = distance;

adjacent->previous = smallest;

}

}

delete adjacentNodes;

}

}

Node\* ExtractSmallest(vector<Node\*>& nodes) {

int size = nodes.size();

if (size == 0) return NULL;

int smallestPosition = 0;

Node\* smallest = nodes.at(0);

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

Node\* current = nodes.at(i);

if (current->distanceFromStart <

smallest->distanceFromStart) {

smallest = current;

smallestPosition = i;

}

}

nodes.erase(nodes.begin() + smallestPosition);

return smallest;

}

vector<Node\*>\* AdjacentRemainingNodes(Node\* node) {

vector<Node\*>\* adjacentNodes = new vector<Node\*>();

const int size = edges.size();

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

Edge\* edge = edges.at(i);

Node\* adjacent = NULL;

if (edge->node1 == node) {

adjacent = edge->node2;

} else if (edge->node2 == node) {

adjacent = edge->node1;

}

if (adjacent && Contains(nodes, adjacent)) {

adjacentNodes->push\_back(adjacent);

}

}

return adjacentNodes;

}

int Distance(Node\* node1, Node\* node2) {

const int size = edges.size();

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

Edge\* edge = edges.at(i);

if (edge->Connects(node1, node2)) {

return edge->distance;

}

}

return -1;

}

bool Contains(vector<Node\*>& nodes, Node\* node) {

const int size = nodes.size();

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

if (node == nodes.at(i)) {

return true;

}

}

return false;

}

void PrintShortestRouteTo(Node\* destination) {

Node\* previous = destination;

cout << "Distance from start: "

<< destination->distanceFromStart << endl;

while (previous) {

cout << previous->id << " ";

previous = previous->previous;

}

cout << endl;

}

vector<Edge\*>\* AdjacentEdges(vector<Edge\*>& Edges, Node\* node);

void RemoveEdge(vector<Edge\*>& Edges, Edge\* edge);

vector<Edge\*>\* AdjacentEdges(vector<Edge\*>& edges, Node\* node) {

vector<Edge\*>\* adjacentEdges = new vector<Edge\*>();

const int size = edges.size();

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

Edge\* edge = edges.at(i);

if (edge->node1 == node) {

cout << "adjacent: " << edge->node2->id << endl;

adjacentEdges->push\_back(edge);

} else if (edge->node2 == node) {

cout << "adjacent: " << edge->node1->id << endl;

adjacentEdges->push\_back(edge);

}

}

return adjacentEdges;

}

void RemoveEdge(vector<Edge\*>& edges, Edge\* edge) {

vector<Edge\*>::iterator it;

for (it = edges.begin(); it < edges.end(); ++it) {

if (\*it == edge) {

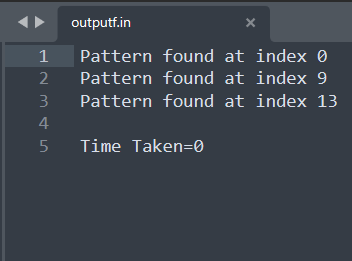
edges.erase(it);

return;

}

}

}



EXPERIMENT-12

AIM:- To implement Naïve string matching algorithm and analyse its complexity.

#include <bits/stdc++.h>

#include<time.h>

using namespace std;

void search(char\* pat, char\* txt)

{

int M = strlen(pat);

int N = strlen(txt);

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

int j;

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

if (txt[i + j] != pat[j])

break;

if (j == M)

cout << "Pattern found at index "

<< i << endl;

}

}

int main()

{

clock\_t start, end;

const double CLK\_TCK = 1000.0;

double t;

char txt[] = "AABAACAADAABAAABAAAA";

char pat[] = "AABA";

start=clock();

search(pat, txt);

end=clock();

t=((double) (end-start))/CLOCKS\_PER\_SEC;

cout<<endl<<"Time Taken="<<t-1;

return 0;

}

