2021

COCSC06(DAA)-LAB FILE

BY: AMOGH GARG
ROLL NUMBER: 2020UCO1688

NETAJI SUBAS UNIVERSITY OF TECHNOLOGY | NEW DELHI

INDEX

SNO	LAB ASSIGNMENT	DATE
1	Write a program for Iterative and Recursive Binary Search	21.10.21
2	Write a program for Merge Sort, Quick Sort	22.10.21
3	Write a program for Minimum Spanning Trees using Kruskal's algorithm and Prim's algorithm	29.11.21
4	Write a program for Floyd-Warshall algorithm.	30.11.21
5	Write a program for Single Source Shortest Path, write a program for Traveling salesman problem	1.12.21
6	Write a program for Optimal Merge Patterns, Write a program for Huffman Coding	2.12.21
7	Write a program for Hamiltonian Problem, Write a program for Strassen's Matrix Multiplication	3.12.21

LAB-1

CODE-ITERATIVE:

```cpp
#include <iostream>
using namespace std;
int main(){
    int a[]={1,3,5,7,9};
    int n=sizeof(a)/sizeof(int);
    for(int i=0;i<n;i++){
       cout<<a[i]<<' ';
    }
    cout<<endl;
    int key;
    cin>>key;
    int s=0,e=n-1,ans=-1; //s is start,e is end,m is mid
    while(s<=e){
       int m=(s+e)/2;
       if(a[m]==key){
           ans=m;
           break;
       }
       else if(a[m]>key){
           e=m-1;
       }
       else if(a[m]<key){
           s=m+1;
       }
    }
    if(ans==-1){
       cout<<"Key not found"<<endl;
    }
    else{
       cout<<"Key Found at index:"<<ans<<endl;
    }
```

CODE-RECURSIVE:

```cpp
#include<bits/stdc++.h>
using namespace std;
//Linear Search
bool LinearSearch(int *lsarr,int i,int k){
    for(int j=0;j<i+1;j++){
        if (lsarr[j]==k){
            return true;
        }
    }
    return false;

}
//Binary Search
bool BinarySearch(int *bsarr,int s,int e,int k){
    if(s<=e){
        int mid=(s+e)/2;
        if(bsarr[mid]==k){
            return mid;
        }
        else if(bsarr[mid]>k){
            BinarySearch(bsarr,s,mid-1,k);
        }
        else{
            BinarySearch(bsarr,mid+1,e,k);
        }

    }
    return -1;
}
```

```cpp
int main(){
    int lsarr[100],bsarr[100],i=0; //Separate arrays->To keep sorted data for Binary Search
    ifstream f;                    //and unsorted for Linear Search
    f.open("inp1.txt");
    int x;
    while(f>>x){
        lsarr[i]=x;
        bsarr[i]=x;
        i++;
    }
    sort(bsarr,bsarr+i+1);       //Sorting Binary Search array
    int k;
    cout<<"Enter the element to search for:";
    cin>>k;
    //Binary Search Result
    if(BinarySearch(bsarr,0,i,k)!= -1){
        cout<<"Binary Search:Element is present at "<<BinarySearch(bsarr,0,i,k)<<endl;
    }
    else{
        cout<<"Binary Search:Element is not present"<<endl;
    }
    //Linear Serch Result
    if(LinearSearch(lsarr,i,k)){
        cout<<"Linear Search:Element is present"<<endl;
    }
    else{
        cout<<"Linear Search:Element is not present"<<endl;
    }
    return 0;
```

OUTPUT-ITERATIVE:

```
PS D:\C++ Fundamentals> .\binary_search.exe
1 3 5 7 9
5
Key Found at index:2
PS D:\C++ Fundamentals> .\binary_search.exe
1 3 5 7 9
10
Key not found
PS D:\C++ Fundamentals>
```

OUTPUT-RECURSIVE:

```
PS D:\NSUT Work\DSA\Programming Excercises-I> .\PE-1.exe
Enter the element to search for:54
Binary Search:Element is present at 1
Linear Search:Element is present
```

LAB-2

CODE-MERGE SORT:

```cpp
//Time Complexity is O(NlogN)
#include <iostream>
using namespace std;
#define ll long long int

void mergetwosortedarray(ll *arr,ll s,ll e){
    int mid=(s+e)/2;
    int i=s;
    int j=mid+1;
    ll *temp= new ll[2000000];
    int k=s;
    while(i<=mid&&j<=e){
        if(arr[i]<arr[j]){
            temp[k++]=arr[i++];
        }
        else{
            temp[k]=arr[j];
            k++;
            j++;
        }
    }
    while(i<=mid){
        temp[k]=arr[i];
        i++;
        k++;
    }
    while(j<=e){
        temp[k]=arr[j];
        j++;
        k++;
    }
    for(ll i=s;i<=e;i++){
        arr[i]=temp[i];//copying temp in arr;
    }
    delete [] temp;
}
void mergesort(ll *arr,ll s,ll e){
    //base case
    if(s>=e){
        return;
    }
    // recursive case
    int mid=(s+e)/2;
    mergesort(arr,s,mid); // 1 2 4
    mergesort(arr,mid+1,e); // 3 5
    mergetwosortedarray(arr,s,e);


}
int main(){
    int n;
    cout<<"Number of elements in Array:";
    cin>>n;
    cout<<"Input Array:";
    ll *arr=new ll[2000000];
    for (int i = 0; i < n; ++i){
        cin>>arr[i];
    }
```

```
    }
    cout<<"Output Array:";
    mergesort(arr,0,n-1);
    for (int i = 0; i < n; ++i){
        cout<<arr[i]<<" ";
    }
    cout<<endl;
    delete []arr;
}
```

CODE-QUICK SORT:

```
//Time Complexity is O(NlogN)
#include <bits/stdc++.h>
using namespace std;
int partition (int arr[], int low, int high){
    int pivot = arr[high]; // pivot->last element
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++){
        if (arr[j] < pivot){
            i++;
            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){
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
void printArray(int arr[], int size){
    int i;
    for (i = 0; i < size; i++){
        cout << arr[i] << " ";
    }
    cout << endl;
}
int main(){
    int arr[] = {34, 27, 90, 76, 1, 18};
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n - 1);
    cout << "Sorted array:";
    printArray(arr, n);
    return 0;
}
```

OUTPUT-MERGE SORT:

```
PS D:\C++ Master Course> .\Merge_Sort.exe
Number of elements in Array:8
Input Array:21 34 16 78 67 90 89 90
Output Array:16 21 34 67 78 89 90 90
PS D:\C++ Master Course>
```

OUTPUT-QUICK SORT:

```
PS D:\NSUT Work\DSA\Programming Excercises-II> .\Quick_Sort.exe
Sorted array:1 18 27 34 76 90
```

LAB-3

CODE-KRUSKALS:

```cpp
#include <bits/stdc++.h>
using namespace std;
class DSU{
    int *parent;
    int *rank;
public:
    //Constructor
    DSU(int n){
        parent = new int[n];
        rank = new int[n];
        for (int i = 0; i < n; i++){
            parent[i] = -1;
            rank[i] = 1;
        }
    }
    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});
    }
    int kruskals_mst(){
        sort(edgelist.begin(), edgelist.end());

        // Initialize the DSU
        DSU s(V);
        int ans = 0;
        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;
            }
        }
        return ans;
    }
};
int main(){
    Graph g(4);
    g.addEdge(0, 1, 1);
    g.addEdge(1, 3, 3);
    g.addEdge(3, 2, 4);
    g.addEdge(2, 0, 2);
    g.addEdge(0, 3, 2);
    g.addEdge(1, 2, 2);
    cout << g.kruskals_mst();
    return 0;
}
```

CODE-PRIMS:

```
//Time Complexity is O(V^2)
#include <bits/stdc++.h>
using namespace std;
#define V 5

int minKey(int key[], bool mstSet[]){
    // Initialize min value
    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 print(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]]<<endl;
}
void prim(int graph[V][V]){
    // Array to store constructed MST
    int parent[V];
    int key[V];
    // To represent set of vertices included in MST
    bool mstSet[V];
    for (int i = 0; i < V; i++){
        key[i] = INT_MAX, mstSet[i] = false;
    }
```

```
30      key[0] = 0;
31      parent[0] = -1;
32      for (int count = 0; count < V - 1; count++){
33          int u = minKey(key, mstSet);
34          mstSet[u] = true;
35          for (int v = 0; v < V; v++)
36              if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
37                  parent[v] = u, key[v] = graph[u][v];
38      }
39      print(parent, graph);
40  }
41  int main(){
42      int graph[V][V]={0};
43      ifstream f;
44      f.open("inp.txt");
45      string line;
46      while(getline(f,line)){
47          vector<int> temp;
48          stringstream lineStream(line);
49          int value;
50          while(lineStream >> value){
51              temp.push_back(value);
52          }
53          graph[temp[0]][temp[1]]=temp[2];
54      }
55      prim(graph);
56      return 0;
57  }
58
```

OUTPUT-KRUSKALS:

```
PS D:\NSUT Work\DSA\Programming Excercises-II> .\Kruskals_Algo.exe
5
```

OUTPUT-PRIMS:

```
PS D:\NSUT Work\DSA\Programming Excercises-II> .\Prims_Algo.exe
Edge    Weight
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   5
```

LAB-4

CODE:

```cpp
#include <bits/stdc++.h>
using namespace std;
#define V 4
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] == INT_MAX)
                cout << "INF"<< "    ";
            else
                cout << dist[i][j] << "  ";
        }
        cout << endl;
    }
}
void floydWarshall(int graph[][V]){
    int dist[V][V], i, j, k;
    for (i = 0; i < V; i++){
        for (j = 0; j < V; j++){
            dist[i][j] = graph[i][j];
        }
    }
    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 (dist[i][j] > (dist[i][k] + dist[k][j]) && (dist[k][j] != INT_MAX && dist[i][k] != INT_MAX)){
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }
    printSolution(dist);
}
int main(){
    int graph[V][V] = { { 0, 5, INT_MAX, 10 },
                        { INT_MAX, 0, 3, INT_MAX },
                        { INT_MAX, INT_MAX, 0, 1 },
                        { INT_MAX, INT_MAX, INT_MAX, 0 } };
    floydWarshall(graph);
    return 0;
}

```

OUTPUT:

```
PS D:\NSUT Work\DSA\Programming Excercises-II> .\Flyod-Warshall.exe
The following matrix shows the shortest distances between every pair of vertices
0       5       8       9
INF     0       3       4
INF     INF     0       1
INF     INF     INF     0
```

LAB-5

CODE-DIJKSTRAS:

```cpp
//Time Complexity is O(V^2)
#include <bits/stdc++.h>
using namespace std;
#define V 5

int minDistance(int dist[], bool sptSet[]){
    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;
}
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;
}
void dijkstra(int graph[V][V], int src){
    int dist[V];
    bool sptSet[V];
    for (int i = 0; i < V; i++){
        dist[i] = INT_MAX, sptSet[i] = false;
    }
    dist[src] = 0;
    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = true;
       // Update dist value of the adjacent vertices of the picked vertex.
       for (int v = 0; v < V; 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];
            }
    }
    printSolution(dist);
}
int main(){
    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}};
    dijkstra(graph, 0);
    return 0;
}

```

CODE-TRAVELLING SALESMAN:

```cpp
#include <bits/stdc++.h>
using namespace std;
#define V 4
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;
}
int main(){
    // Graph
    int graph[][V] = { { 0, 10, 15, 20 },
```

OUTPUT-DIJKSTRAS:

```
PS D:\NSUT Work\DSA\Programming Excercises-II> .\Dijkstra.exe
Vertex   Distance from Source
0                0
1                2
2                5
3                6
4                7
```

OUTPUT-TRAVELLING SALESMAN:

```
PS D:\NSUT Work\DSA\Programming Excercises-II> .\Travelling_Salesman.exe
80
```

LAB-6

CODE-OPTIMAL MERGE:

```cpp
#include <bits/stdc++.h>
using namespace std;
int minComputation(int size, int files[]){
    priority_queue<int, vector<int>, greater<int> > pq;
    for (int i = 0; i < size; i++) {
        pq.push(files[i]);
    }
    int count = 0;

    while (pq.size() > 1) {
        int first_smallest = pq.top();
        pq.pop();
        int second_smallest = pq.top();
        pq.pop();

        int temp = first_smallest + second_smallest;
        count += temp;
        pq.push(temp);
    }
    return count;
}
int main(){
    int n = 6;

    int files[] = { 2, 3, 4, 5, 6, 7 };
    cout << "Minimum Computations = "
        << minComputation(n, files);

    return 0;
}

```

CODE-HUFFMAN ENCODING:

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

class node{
public:
    char data;
    unsigned freq;
    node * left;
    node * right;

    node(char d, unsigned f){
        data=d;
        freq=f;
        right=NULL;
        left=NULL;
    }
};
// For Min Heap
class myComparator{
public:
    int operator()(node* p1,node* p2){
        return p1->freq>p2->freq;
    }
};
void printc(node* root,string str){
    if (root==NULL)
        return;
    if (root->data != '$')
        cout<<root->data<<": "<<str<<endl;
    printc(root->left,str + "0");
    printc(root->right,str + "1");
}
void Encoding(char data[],int freq[],int size){
    node *left, *right, *top;
    // Built in Min Heap
    priority_queue<node*, vector<node*>,myComparator> minHeap;
```

```
    // Built in Min Heap
    priority_queue<node*, vector<node*>,myComparator> minHeap;
    for (int i=0;i<size;++i){
        minHeap.push(new node(data[i], freq[i]));
    }
    while (minHeap.size() != 1) {
        left = minHeap.top();
        minHeap.pop();
        right = minHeap.top();
        minHeap.pop();
        top = new node('$', left->freq + right->freq);
        top->left = left;
        top->right = right;
        minHeap.push(top);
    }
    printc(minHeap.top(),"");
}
int main(){
    char a[] = {'a','b','c','d','e','f'};
    int freq[] = {5,9,12,13,16,45};
    int n= sizeof(a)/sizeof(a[0]);
    Encoding(a,freq,n);
    return 0;
}
```

OUTPUT-OPTIMAL MERGE:

```
PS D:\NSUT Work\DSA\Programming Excercises-II> .\Optimal_Merge.exe
Minimum Computations = 68
```

OUTPUT-HUFFMAN ENCODING:

```
PS D:\NSUT Work\DSA\Programming Excercises-I> .\PE-16.exe
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111
```

LAB-7

CODE-HAMILTONIAN PROBLEM:

```
#include <bits/stdc++.h>
#define N 5
using namespace std;
void displaytheSolution(int path[]);
bool isSafe(int n, bool g[N][N], int path[], int pos) {
   if (g [path[pos-1]][n] == 0)
      return false;
   for (int i = 0; i < pos; i++)
      if (path[i] == n)
         return false;
   return true;
}
bool hamiltonianCycle(bool g[N][N], int path[], int pos) {
   //If all vertices are included in Hamiltonian Cycle
   if (pos == N) {
      if (g[ path[pos-1] ][ path[0] ] == 1)
         return true;
      else
         return false;
   }
   for (int n = 1; n < N; n++) {
      if (isSafe(n, g, path, pos)) { //Check if this vertex can be added to Hamiltonian Cycle
         path[pos] = n;
         //recur to construct rest of the path
         if (hamiltonianCycle (g, path, pos+1) == true)
            return true;
         path[pos] = -1; //remove vertex if it doesn't lead to the solution
      }
   }
   return false;
}
bool hamCycle(bool g[N][N]) {
   int *path = new int[N];
   for (int i = 0; i < N; i++)
   path[i] = -1;
   //put vertex 0 as the first vertex in the path. If there is a Hamiltonian Cycle, then the path can be started from any point
```

```
bool hamCycle(bool g[N][N]) {
   int *path = new int[N];
   for (int i = 0; i < N; i++)
   path[i] = -1;
   //put vertex 0 as the first vertex in the path. If there is a Hamiltonian Cycle, then the path can be started from any point
   //of the cycle as the graph is undirected
   path[0] = 0;
   if (hamiltonianCycle(g, path, 1) == false) {
      cout<<"\nCycle does not exist"<<endl;
      return false;
   }
   displaytheSolution(path);
   return true;
}
void displaytheSolution(int p[]) {
   cout<<"Cycle Exists:";
   cout<<" Following is one Hamiltonian Cycle \n"<<endl;
   for (int i = 0; i < N; i++)
   cout<<p[i]<<" ";
   cout<< p[0]<<endl;
}
int main() {
   bool g[N][N] = {{0, 1, 0, 1, 1},
      {0, 0, 1, 1, 0},
      {0, 1, 0, 1, 1},
      {1, 1, 1, 0, 1},
      {0, 1, 1, 0, 0},
   };
   hamCycle(g);
   return 0;
}
```

CODE-STRASSENS MULTIPLICATION:

```cpp
#include<iostream>
using namespace std;
double a[4][4];
double b[4][4];

void insert(double x[4][4])
{
    double val;
    for(int i=0;i<4;i++)
    {
        for(int j=0;j<4;j++)
        {
            cin>>val;
            x[i][j]=val;
        }
    }
}
double cal11(double x[4][4]){
    return (x[1][1] * x[1][2])+ (x[1][2] * x[2][1]);
}
double cal21(double x[4][4]){
    return (x[3][1] * x[4][2])+ (x[3][2] * x[4][1]);
}

double cal12(double x[4][4]){
    return (x[1][3] * x[2][4])+ (x[1][4] * x[2][3]);
}

double cal22(double x[4][4]){
    return (x[2][3] * x[1][4])+ (x[2][4] * x[1][3]);
}

int main(){
    double a11,a12,a22,a21,b11,b12,b21,b22,a[4][4],b[4][4];
    double p,q,r,s,t,u,v,c11,c12,c21,c22;
    //insert values in the matrix a
```

```cpp
int main(){
    double a11,a12,a22,a21,b11,b12,b21,b22,a[4][4],b[4][4];
    double p,q,r,s,t,u,v,c11,c12,c21,c22;
    //insert values in the matrix a
    cout<<"\n a: \n";
    insert(a);
    //insert values in the matrix a
    cout<<"\n b: \n";
    insert(b);

    //dividing single 4x4 matrix into four 2x2 matrices
    a11=cal11(a);
    a12=cal12(a);
    a21=cal21(a);
    a22=cal22(a);
    b11=cal11(b);
    b12=cal12(b);
    b21=cal21(b);
    b22=cal22(b);

    //assigning variables acc. to strassen's algo
    p=(a11+a22)*(b11+b22);
    q=(a21+a22)*b11;
    r=a11*(b12-b22);
    s=a22*(b21-b11);
    t=(a11+a12)*b22;
    u=(a11-a21)*(b11+b12);
    v=(a12-a22)*(b21+b22);

    //outputting the final matrix
cout<<"\n final matrix";
    cout<<"\n"<<p+s-t+v<<" "<<r+t;
    cout<<"\n"<<q+s<<" "<<p+r-q+u;
    return 0;
}
```

OUTPUT-HAMILTONIAN PROBLEM:

```
PS D:\NSUT Work\DSA\Programming Excercises-II> .\Hamiltonian_Problem.exe
Cycle Exists: Following is one Hamiltonian Cycle

0 4 1 2 3 0
```

OUTPUT-STRASSENS MULTIPLICATION:

```
PS D:\NSUT Work\DSA\Programming Excercises-II> .\Strassens_Multiplication.exe

 a:
1 5 3 7
4 2 6 2
7 2 7 2
9 2 6 2

 b:
5 4 2 6
4 6 6 1
5 4 2 6
7 1 4 7

 final matrix
2176 2072
2.77174e+262 -7.25272e+262
```