Name: Prashant Kumar
Roll No.: 222CS2097
M.Tech.: 1st Sem
Assignment: APL 2

Q.4. Average case analysis for Sorting Algorithms

For each of the data formats: random, reverse ordered, and nearly sorted, run your program say **SORTTEST** for all combinations of sorting algorithms and data sizes and complete each of the following tables. When you have completed the tables, analyze your data and determine the asymptotic behavior of each of the sorting algorithms for each of the data types *(i)* ***Random data, (ii) Reverse Ordered Data, (iii) Almost Sorted Data*** **and (iv)** ***Highly Repetitive Data*** . select the suitable no of elements for the analysis that supports your program.

(i) Random Data

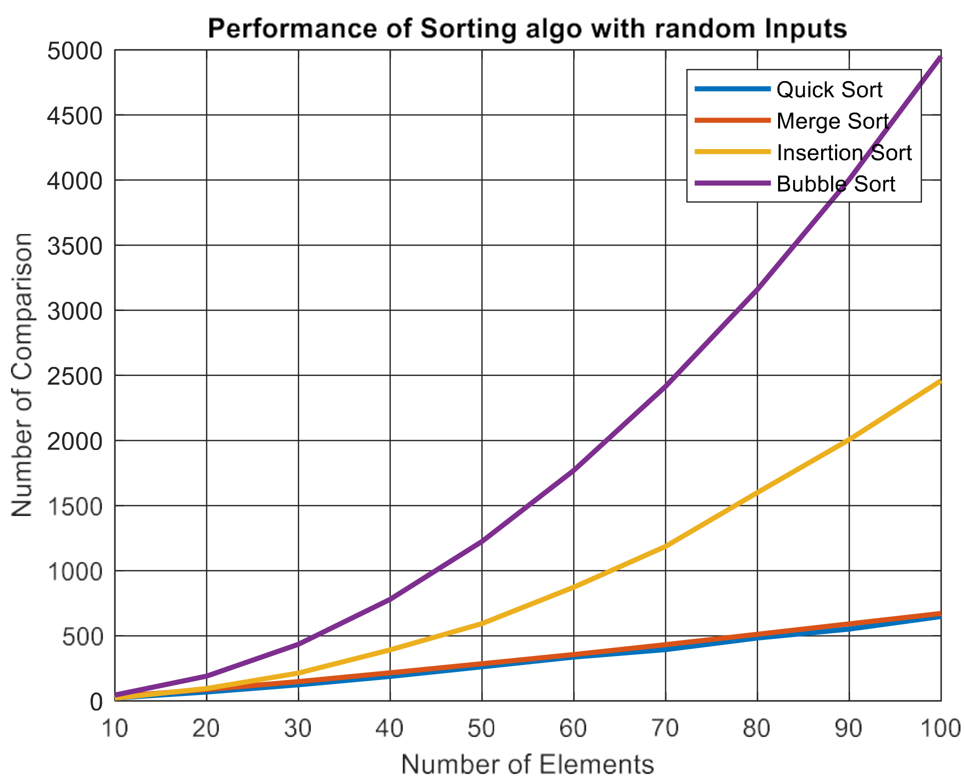


Fig. 4.1: Analysing performance of different sorting algo on random input

Observation: Comparing Bubble sort, Insertion sort, Merge sort and Quick sort on random data set. The Y-axis represent the comparison needed and X-axis represent the number of data set.

We can observe that Bubble sort takes quadratic time . Insertion Sort takes less time in all cases than bubble sort and more time than Quicksort and Merge sort. Quicksort and Merge sort takes significant less time than bubble sort and Insertion sort because they both have Average Time Complexity as O(nlogn).

(ii) Reverse Ordered Data

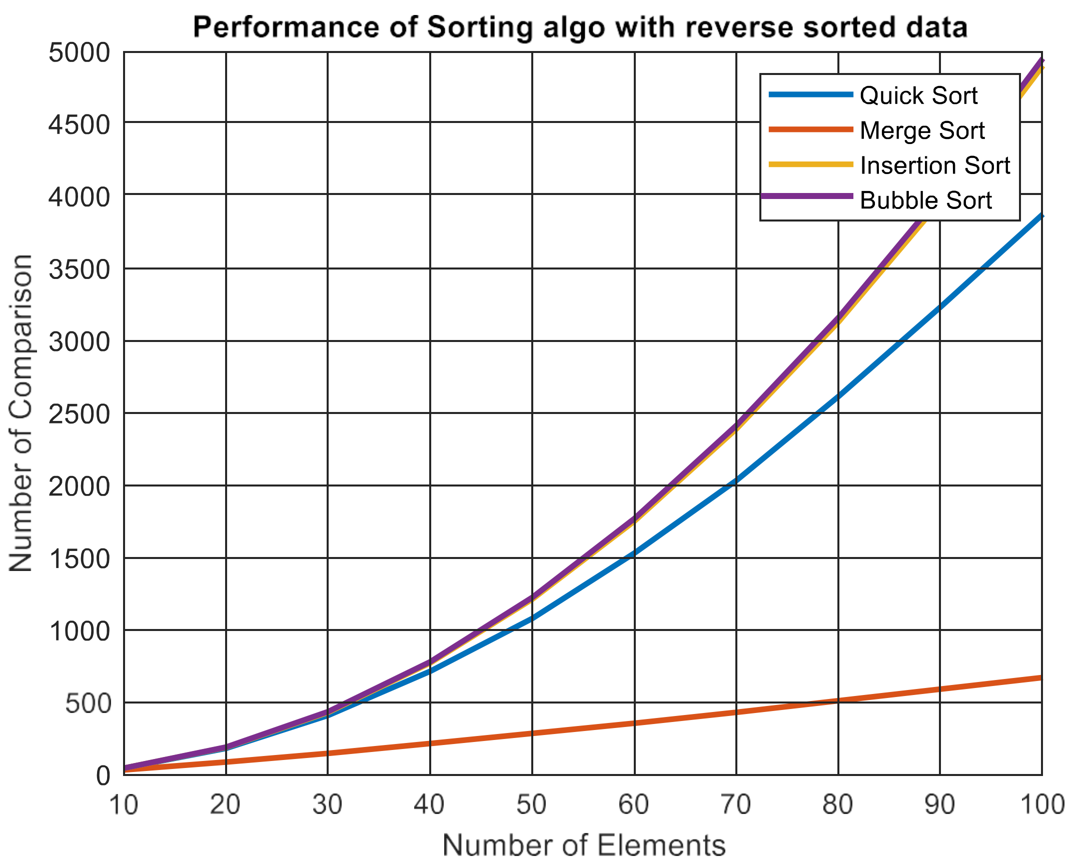


Fig. 4.2: Analysing performance of different sorting algo on reverse sorted input

Observation: Comparing Bubble sort, Insertion sort, Merge sort and Quick sort on reverse dataset. The Y-axis represent the comparison needed and X-axis represent the number of data set.

In the above graph we observe that Insertion sort and Bubble sort both perform in quadratic time as every iteration we need to shift all the elements of the subarray, the time complexity becomes same as that of bubble sort. Quicksort takes slightly less no. of comparisons than Bubble sort and Insertion sort. Merge sort perform better for reverse dataset as it has worst Time Complexity as O(nlogn).

(iii) Almost Sorted Data

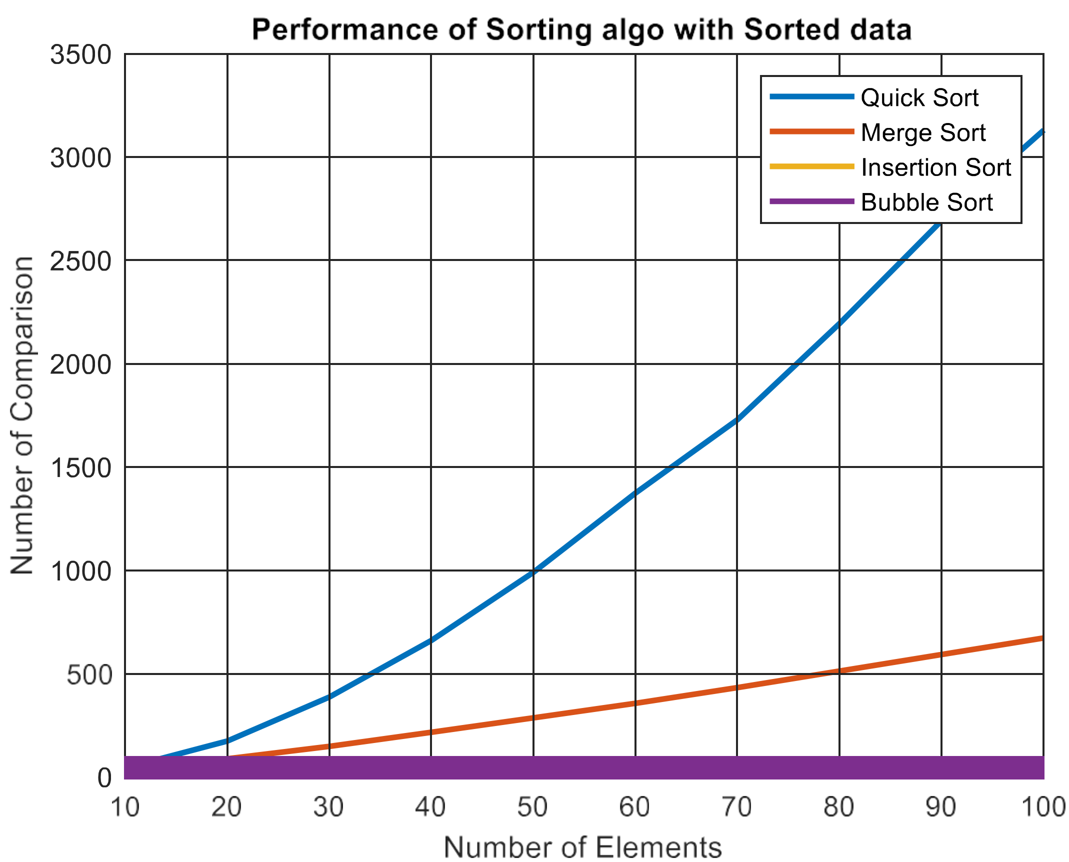


Fig. 4.3: Analysing performance of different sorting algo on sorted input

Observation: Comparing Bubble sort, Insertion sort, Merge sort and Quick sort on sorted dataset. The Y-axis represent the comparison needed and X-axis represent the number of data set.

In the above graph we observe that Quick sort perform in quadratic time as every iteration we will have pivot element at right place but have iterate for rest element, Merge sort and Insertion sort will complete in O(n), Merge sort will take O(nlogn) time as always.

(iv) Highly Repetitive Data

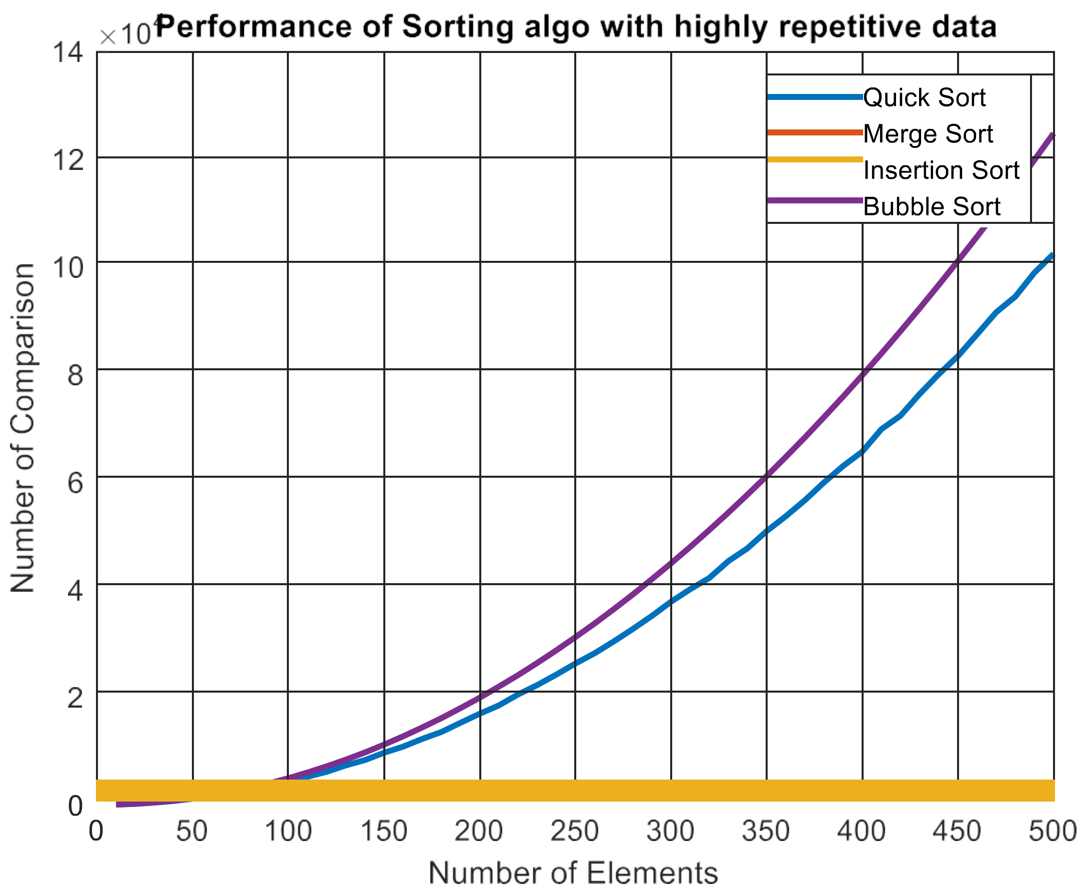


Fig. 4.4: Analysing performance of different sorting algo on repetitive data input

Observation: Comparing Bubble sort, Insertion sort, Merge sort and Quick sort on highly repeated dataset. The Y-axis represent the comparison needed and X-axis represent the number of data set.

In above graph shows we observe that for arrays with repetitive data Merge sort and Insertion sort performs the best with least number of comparisons. Bubble sort takes quadratic time and Quick sort take less time than bubble sort but significantly more than Insertion sort.

Code:

// Bubble sort

```
function [comp] = bs1(arr,n)

comp=0;
for i=1:n
    for j=1:n-i
        comp=comp+1;
        if arr(j)>arr(j+1)
           // swap logic
           tt=arr(j);
           arr(j)=arr(j+1);
           arr(j+1)=tt;
        end
    end
end
end
```

// Merge sort

```
function [comp] = mergeSort(x,n)
comp=0;
[x,comp] = mergeSorti(x,1,n, comp);
end

function [x, comp] = mergeSorti(x,ll,uu,comp)
if ll<uu
```

```
    mid= floor((ll+uu)/2);
    [x, comp] = mergeSorti(x,ll,mid,comp);
    [x, comp] = mergeSorti(x,mid+1,uu,comp);
    [x, comp] = merge(x,ll,mid, uu, comp);
end
end


function [x, comp] = merge(x, ll, mid, uu, comp)
n1 = mid-ll+1;
n2=uu-mid;

arr1= [];
arr2= [];

for i=1:n1
   arr1(i)=x(ll+i-1);
end
for j=1:n2
   arr2(j)=x(mid+j);
end

arr1(n1+1) = inf;
arr2(n2+1) = inf;
i=1;
j=1;

for k = ll : uu
    comp=comp+1;
    if arr1(i) <= arr2(j)
        x(k) = arr1(i);
        i = i + 1;
    else
        x(k) = arr2(j);
        j = j + 1;
    end
end
end
```

// Insertion sort

```
function [comp] = is(arr,n)
comp=0;
for i=1:n
    key=arr(i);
    j=i-1;
    %comp=comp+1;
    while j>0 && arr(j)>key
        comp=comp+1;
        arr(j+1)=arr(j);
        j=j-1;
    end
    arr(j+1)=key;
end
end
```

// Quick sort

```
function [comp] = qs(arr,n)
comp=0;
[arr,comp] = quicksorti(arr,1,n, comp);
end

function [arr, comp] = quicksorti(arr,ll,uu,comp)
if ll<uu
    [arr, pi, comp] = partition(arr,ll,uu, comp);
    [arr, comp] = quicksorti(arr,ll,pi - 1,comp);
    [arr, comp] = quicksorti(arr,pi + 1,uu,comp);

end
end

function [arr, pi, comp] = partition(arr,ll,uu, comp)
pivot=arr(uu);
i=ll-1;
for j=ll:uu-1
    comp = comp+1;
    if arr(j)<pivot

        i=i+1;
```

```
        temp=arr(j);
        arr(j)=arr(i);
        arr(i)=temp;
    end
end
i=i+1;
temp2=arr(uu);
arr(uu)=arr(i);
arr(i)=temp2;
pi=i;
end
```

// Random Data Input

```
clear all
noelt = zeros(1,10);
qscomp= zeros(1,10);
mscomp= zeros(1,10);
iscomp= zeros(1,10);
bscomp= zeros(1,10);
comp=0;
k=1;

for n=10:10:100
    noelt(k)=n;

    c1=0;c2=0;c3=0;c4=0;
    for z=1:50
        arr=round(rand(1,n)*100);
        % calling
        c1= c1+qs(arr,n);
        c2= c2+ mergeSort(arr,n);
        c3= c3+is(arr,n);
        c4= c4+bs1(arr,n);
    end
    qscomp(k)=c1/50;
    mscomp(k)=c2/50;
    iscomp(k)=c3/50;
    bscomp(k)=c4/50;
    k=k+1;
end
```

```
plot(noelt, qscomp,noelt, mscomp,noelt,iscomp,noelt,bscomp,
'Linewidth',2)
legend('Quick Sort','Merge Sort','Insertion Sort','Bubble
Sort')
title('Performance of Sorting algo with random Inputs')
xlabel('Number of Elements')
ylabel('Number of Comparison')
grid on
```

// Reverse Sorted Input

```
nelt = zeros(1,10);
qscomp= zeros(1,10);
mscomp= zeros(1,10);
iscomp= zeros(1,10);
bscomp= zeros(1,10);
comp=0;
k=1;

for n=10:10:100
    nelt(k)=n;

    c1=0;c2=0;c3=0;c4=0;
    for z=1:50
        B=round(rand(1,n)*100);
        A=sort(B);
        arr=flip(A);
        % calling
        c1= c1+qs(arr,n);
        c2= c2+ mergeSort(arr,n);
        c3= c3+is(arr,n);
        c4= c4+bs1(arr,n);
    end
    qscomp(k)=c1/50;
    mscomp(k)=c2/50;
    iscomp(k)=c3/50;
    bscomp(k)=c4/50;
    k=k+1;
end
plot(nelt, qscomp,nelt,  mscomp, nelt, iscomp, nelt,
bscomp, 'Linewidth', 2)
```

```
legend('Quick Sort','Merge Sort','Insertion Sort','Bubble
Sort')
title('Performance of Sorting algo with reverse sorted
data')
xlabel('Number of Elements')
ylabel('Number of Comparison')
grid on
```

// Almost Sorted Input

```
nelt = zeros(1,10);
qscomp= zeros(1,10);
mscomp= zeros(1,10);
iscomp= zeros(1,10);
bscomp= zeros(1,10);
comp=0;
k=1;

for n=10:10:100
    nelt(k)=n;

    c1=0;c2=0;c3=0;c4=0;
    for z=1:50
        B=round(rand(1,n)*100);
        arr=sort(B);
        % calling
        c1= c1+qs(arr,n);
        c2= c2+ mergeSort(arr,n);
        c3= c3+is(arr,n);
        c4= c4+bs2(arr,n);
    end

    qscomp(k)=c1/50;
    mscomp(k)=c2/50;
    iscomp(k)=c3/50;
    bscomp(k)=c4/50;
    k=k+1;
end

plot(nelt, qscomp,nelt, mscomp, nelt, iscomp, nelt, bscomp,
'Linewidth', 2)
```

```
legend('Quick Sort','Merge Sort','Insertion Sort','Bubble
Sort')
title('Performance of Sorting algo with Sorted data')
xlabel('Number of Elements')
ylabel('Number of Comparison')
grid on
```

// Highly Repeated Input

```
nelt = zeros(1,10);
qscomp= zeros(1,10);
mscomp= zeros(1,10);
iscomp= zeros(1,10);
bscomp= zeros(1,10);
comp=0;
k=1;

for n=10:10:500
    nelt(k)=n;
    c1=0;c2=0;c3=0;c4=0;
    for z=1:50
        arr=round(rand(1,n)*100);
        for s=1:n
            if arr(s)<90
               arr(s)=50;
           end
        end
        % calling
        c1= c1+ qs(arr,n);
        c2= c2+ mergeSort(arr,n);
        c3= c3+ is(arr,n);
        c4= c4+ bs1(arr,n);
    end

    qscomp(k)=c1/50;
    mscomp(k)=c2/50;
    iscomp(k)=c3/50;
    bscomp(k)=c4/50;
    k=k+1;
end
```

```
plot(nelt, qscomp, nelt, mscomp, nelt, iscomp, nelt,
bscomp, 'Linewidth',2)
legend('Quick Sort','Merge Sort','Insertion Sort','Bubble
Sort')
title('Performance of Sorting algo with highly repetitive
data')
xlabel('Number of Elements')
ylabel('Number of Comparison')
grid on
```

Q.5. Quick Select

Use the **QUICK SELECT** algorithm to find **3rd largest element** in an array of n integers. Analyze the performance of **QUICK SELECT** algorithm for the different instance of size 50 to 500 element. Record your observation with the *number of comparison made* vs. *instance*.

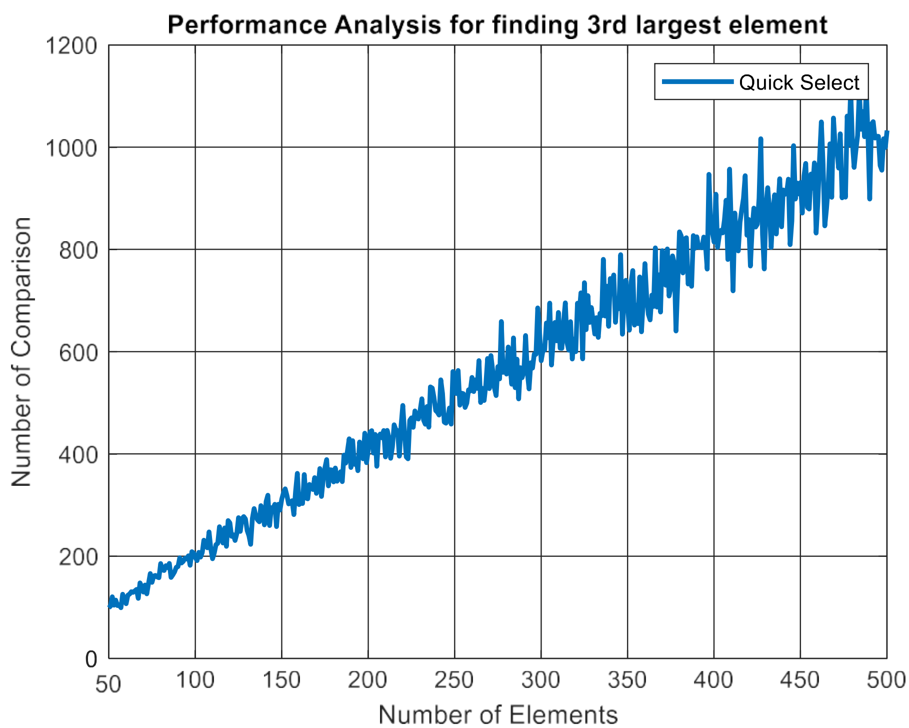


Fig. 5.1: Performance analysis of quick select for finding 3rd largest element in array

Observation: Quick select uses quick sort method of partitioning and then working on the side where the required element is present. It has worst time complexity of $O(n^2)$. The above graph represent the number of comparison required with the corresponding number of element in array. On the average the algorithm follows a linear path. In the above figure, the slope of the linear path is approximately equal to 2.

Code:

```
nelt = zeros(1,10);
qselectcomp= zeros(1,10);
bubbletcomp= zeros(1,10);
comp=0;
k=1;

for n=50:500
    nelt(k)=n;

    c1=0;
    for i=1:30
        arr=round(rand(1,n)*100);
        % calling
        c1= c1+quickSelect(arr,n);
    end
    qselectcomp(k)=c1/30;
    k=k+1;
    comp=0;
end
plot(nelt, qselectcomp, 'Linewidth',2)
legend('Quick Select')
title('Performance Analysis for finding 3rd largest
element')
xlabel('Number of Elements')
ylabel('Number of Comparison')
grid on
```

// Quick select

```
function [comp] = quickSelect(arr,n)
comp=0;
```

```
pos=n-2;
[arr,comp] = quicksorti(arr,1,n, comp, pos);
end

function [arr, comp] = quicksorti(arr,ll,uu,comp, pos)
if ll<uu
    [arr, pi, comp] = partition(arr,ll,uu, comp);
    if pi==pos
        return
    elseif pi>pos
        [arr, comp] = quicksorti(arr,ll,pi - 1,comp, pos);
    else
        [arr, comp] = quicksorti(arr,pi + 1,uu,comp, pos);
    end
end
end

function [arr, pi, comp] = partition(arr,ll,uu, comp)
pivot=arr(uu);
i=ll-1;
for j=ll:uu-1
    comp = comp+1;
    if arr(j)<pivot

        i=i+1;
        temp=arr(j);
        arr(j)=arr(i);
        arr(i)=temp;
    end
end
i=i+1;
temp2=arr(uu);
arr(uu)=arr(i);
arr(i)=temp2;
pi=i;
end
```

Q.6. Iterative Binary Search

Write programs to implement recursive and iterative versions of binary search and compare the performance. For an appropriate size of n=20(say), have each algorithm find every element in the set. Then try all n+1 possible unsuccessful search. The performances, of these versions of binary search are to be reported graphically with your observations.

For successful search:

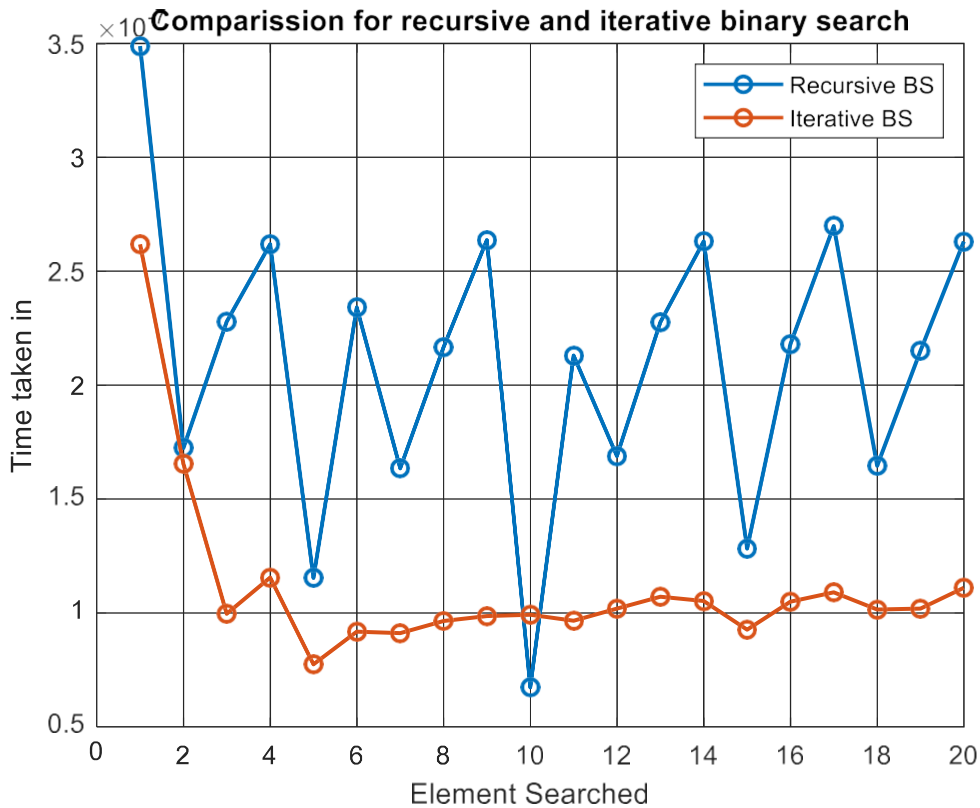


Fig. 6.1: Performance analysis of recursive and iterative binary search

For unsuccessful search :

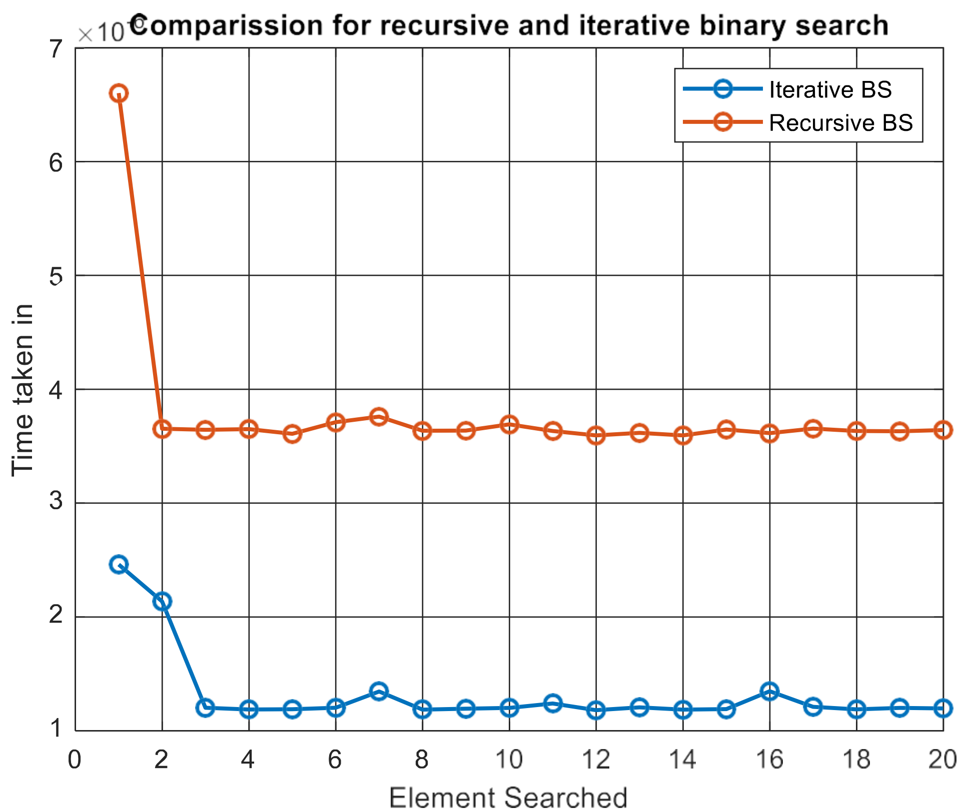


Fig. 6.2: Performance analysis of recursive and iterative binary search

Observation: In the given graph the performance analysis is done on basis of time. y axis denote time required to search and x axis shows the item which is being searched in a sorted array from 1 to 20. We can observe that iterative algorithm perform better than recursive since recursive has overload for calling it again and again.

Code:

// Recursive binary search

```
function index = recursiveBinarySearch(arr,low,high,k)
if low > high
```

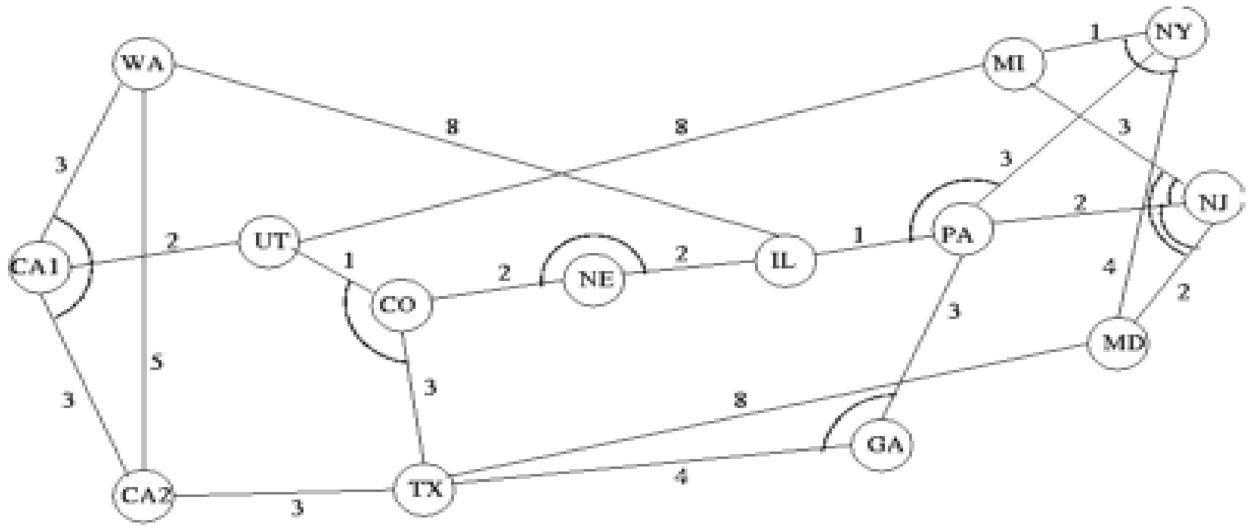
```
    index=0;
else
    mid = floor((high+low)/2);
    if arr(mid) < k
        low = mid+1;
        index = recursiveBinarySearch(arr,low,high,k);
    elseif arr(mid) > k
        high = mid-1;
        index = recursiveBinarySearch(arr,low,high,k);
    else
        index = mid;
    end
end
end
```

// Iterative binary search

```
function [index] = iterativeBinarySearch(A, n, num)
    left = 1;
    right = n;
    flag = 0;
    while left < right
        mid = floor((left + right) / 2);
        if A(mid) == num
            index = mid;
            flag = 1;
            break;
        elseif A(mid) < num
            left = mid + 1;
        else
            right = mid - 1;
        end
    end
    if flag == 0
        index = -1;
    end
end

arr = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20];
a=zeros(1,21);
recursive_time=zeros(1,21);
```

```
iterative_time=zeros(1,21);

k=1;

for i=1:21
    a(k)=i;
    tic
    pos1 = recursiveBinarySearch(arr,1,20,i);
    recursive_time(k)=toc;

    tic
    pos2=iterativeBinarySearch(arr,20,i);
    iterative_time(k)=toc;
    k=k+1;
end

plot(a, recursive_time, a, iterative_time,'Linewidth',2)
legend('Recursive BS','Iterative BS')
title('Performance Analysis of Recursive and Iterative
Binary Search')
xlabel('Element Searched')
ylabel('Time taken in second')
grid on
```

Q.7. Minimum Spanning Tree



Write a program obtain minimum cost spanning tree the above NSF network using Prim's algorithm, Kruskal's algorithm and Boruvka's algorithm. Use the appropriate data structure to store the computed spanning tree for another application (broadcasting a message to all nodes from any source node).

Prim's Algorithm

```
Edges included in minimum spanning tree are
NA->CA1  wt = 3
CA1->CA2  wt = 3
CA1->UT  wt = 2
CO->TX  wt = 3
UT->CO  wt = 1
CO->NE  wt = 2
PA->GA  wt = 3
IL->PA  wt = 1
NJ->MI  wt = 3
MI->NY  wt = 1
PA->NJ  wt = 2
NJ->MD  wt = 2
NE->IL  wt = 2
```

Fig. 7.1 : Output using Prim's algorithm

Code :

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

#define V 14  //No of vertices

int selectMinVertex(vector<int>& value,vector<bool>& setMST)
{
        int minimum = INT_MAX;
        int vertex;
        for(int i=0;i<V;++i)
        {
                if(setMST[i]==false && value[i]<minimum)
                {
                        vertex = i;
                        minimum = value[i];
```

```
        }
    }
    return vertex;
}

void findMST(int graph[V][V], string node_name[V])
{
    int parent[V];
    vector<int> value(V,INT_MAX);
    vector<bool> setMST(V,false);

    //Assuming start point as Node-0
    parent[0] = -1; //Start node has no parent
    value[0] = 0;  //start node has value=0 to get picked 1st

    for(int i=0;i<V-1;++i)
    {
        //Select best Vertex by applying greedy method
        int U = selectMinVertex(value,setMST);
        setMST[U] = true;

        for(int j=0;j<V;++j)
        {
            if(graph[U][j]!=0 && setMST[j]==false && graph[U][j]<value[j])
            {
                value[j] = graph[U][j];
                parent[j] = U;
            }
        }
    }
    //Print MST
    cout<<"Edges included in minimum spanning tree are"<<endl;
    for(int i=1;i<V;++i){
        cout<<node_name[parent[i]]<<"->"<<node_name[i]<<"  wt = "<<
        graph[parent[i]][i]<< "\n";
    }
}

int main()
{
    string node_name[V]= {"NA","CA1","CA2","UT","TX","CO","NE",
                          "GA","PA","MI","NY","NJ","MD","IL"};
    int graph[V][V]  =  { {0, 3, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8},
                          {3, 0, 3, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                          {5, 3, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0},
```

```
                {0, 2, 0, 0, 0, 1, 0, 0, 0, 8, 0, 0, 0, 0},
                {0, 0, 3, 0, 0, 3, 0, 4, 0, 0, 0, 0, 8, 0},
                {0, 0, 0, 1, 3, 0, 2, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 2},
                {0, 0, 0, 0, 4, 0, 0, 0, 3, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 3, 2, 0, 1},
                {0, 0, 0, 8, 0, 0, 0, 0, 0, 0, 1, 3, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 0, 3, 1, 0, 0, 4, 0},
                {0, 0, 0, 0, 0, 0, 0, 0, 2, 3, 0, 0, 2, 0},
                {0, 0, 0, 0, 8, 0, 0, 0, 0, 0, 4, 2, 0, 0},
                {8, 0, 0, 0, 0, 0, 2, 0, 1, 0, 0, 0, 0, 0}};

    findMST(graph,node_name);
    return 0;
}
```

Kruskal's Algorithm

```
Enter 'from' to and weight for edges
0 1 3
0 2 5
0 7 8
1 3 2
1 2 3

2 5 3
3 4 1
3 10 8
4 5 3
4 6 2

5 8 4
5 13 8
6 7 2
7 9 1
8 9 3

9 11 3
9 12 2
10 11 1
10 12 3
11 13 4

12 13 2
MST formed is
from: GA  to: MI  wt: 1
from: NY  to: NJ  wt: 1
from: UT  to: TX  wt: 1
from: MD  to: IL  wt: 2
from: MI  to: MD  wt: 2
from: NE  to: GA  wt: 2
from: TX  to: NE  wt: 2
from: CA1  to: UT  wt: 2
from: NY  to: MD  wt: 3
from: TX  to: CO  wt: 3
from: CA2  to: CO  wt: 3
from: NA  to: CA1  wt: 3
from: PA  to: MI  wt: 3
```

Fig. 7.2 : Output using Kruskal's algorithm

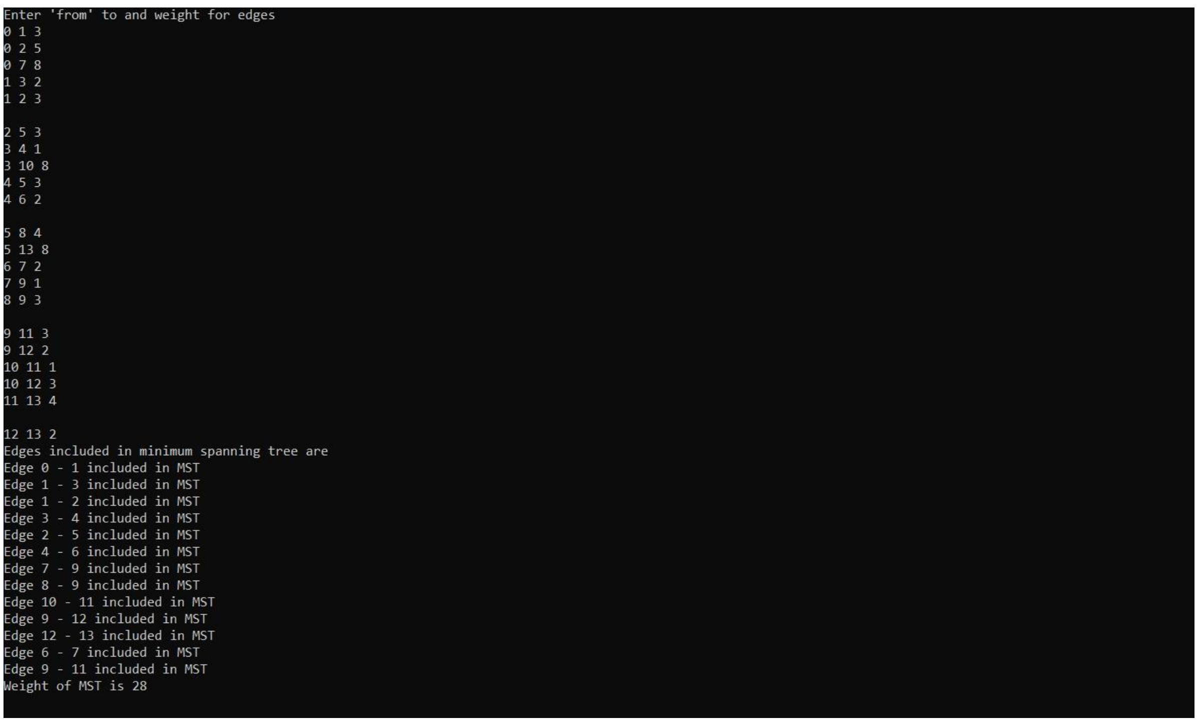
Code :

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

struct node {
	int parent;
	int rank;
};
struct Edge {
	int src;
	int dst;
	int wt;
};

vector<node> dsuf;
vector<Edge> mst;
//FIND operation
int find(int v)
{
	if(dsuf[v].parent==-1)
		return v;
	return dsuf[v].parent=find(dsuf[v].parent);
}

void union_op(int fromP,int toP)
{
	if(dsuf[fromP].rank > dsuf[toP].rank)
		dsuf[toP].parent = fromP;
	else if(dsuf[fromP].rank < dsuf[toP].rank)
		dsuf[fromP].parent = toP;
	else
	{
		dsuf[fromP].parent = toP;
		dsuf[toP].rank +=1;
	}
}

bool comparator(Edge a,Edge b)
{
	return a.wt < b.wt;
}

void Kruskals(vector<Edge>& edge_List,int V,int E)
{
```

```
        sort(edge_List.begin(),edge_List.end(),comparator);

        int i=0,j=0;
        while(i<V-1 && j<E)
        {
                int fromP = find(edge_List[j].src);
                int toP = find(edge_List[j].dst);

                if(fromP == toP)
                {       ++j;    continue;       }

                //UNION operation
                union_op(fromP,toP); //UNION of 2 sets
                mst.push_back(edge_List[j]);
                ++i;
        }
}
void printMST(vector<Edge>& mst,int V)
{
        string node_name[V]= {"NA","CA1","CA2","UT","TX","CO","NE",
                                "GA","PA","MI","NY","NJ","MD","IL"};
        cout<<"MST formed is\n";
        for(int i=0; i<V-1; i++)
        {
                cout<<"from: "<<node_name[mst[i].src]<<"  to: "<<node_name[mst[i].dst]<<"
                        wt: "<<mst[i].wt<<"\n";
   }
}

int main()
{
        int E;
        int V;
        E=21; V=14;


        dsuf.resize(V);//Mark all vertices as separate subsets with only 1 element
        for(int i=0;i<V;++i)    //Mark all nodes as independent set
        {
                dsuf[i].parent=-1;
                dsuf[i].rank=0;
        }

        vector<Edge> edge_List;         //Adjacency list
        Edge temp;
```

```
	cout<<"Enter 'from' to and weight for edges"<<endl;
	for(int i=0;i<E;++i)
	{
		int from,to,wt;
		cin>>from>>to>>wt;
		temp.src = from;
		temp.dst = to;
		temp.wt = wt;
		edge_List.push_back(temp);
	}

	Kruskals(edge_List,V,E);
	printMST(mst,V);

	return 0;
}
```

Boruvka’s Algorithm



```
Enter 'from' to and weight for edges
0 1 3
0 2 5
0 7 8
1 3 2
1 2 3

2 5 3
3 4 1
3 10 8
4 5 3
4 6 2

5 8 4
5 13 8
6 7 2
7 9 1
8 9 3

9 11 3
9 12 2
10 11 1
10 12 3
11 13 4

12 13 2
Edges included in minimum spanning tree are
Edge 0 - 1 included in MST
Edge 1 - 3 included in MST
Edge 1 - 2 included in MST
Edge 3 - 4 included in MST
Edge 2 - 5 included in MST
Edge 4 - 6 included in MST
Edge 7 - 9 included in MST
Edge 8 - 9 included in MST
Edge 10 - 11 included in MST
Edge 9 - 12 included in MST
Edge 12 - 13 included in MST
Edge 6 - 7 included in MST
Edge 9 - 11 included in MST
Weight of MST is 28
```

Fig. 7.3 : Output using Boruvka’s algorithm

Code :

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

struct Edge
{
	int src, dest, weight;
};

struct Graph
{
	int V, E;
	Edge* edge;
};

// A structure to represent a subset for union-find
struct subset
{
	int parent;
	int rank;
};

int find(struct subset subsets[], int i);
void Union(struct subset subsets[], int x, int y);

void boruvkaMST(struct Graph* graph)
{
	int V = graph->V, E = graph->E;
	Edge *edge = graph->edge;

	struct subset *subsets = new subset[V];
	int *cheapest = new int[V];

	// Create V subsets with single elements
	for (int v = 0; v < V; ++v)
	{
		subsets[v].parent = v;
		subsets[v].rank = 0;
		cheapest[v] = -1;
	}

	int numTrees = V;
	int MSTweight = 0;

	while (numTrees > 1)
```

```
{
	for (int v = 0; v < V; ++v)
	{
		cheapest[v] = -1;
	}
	for (int i=0; i<E; i++)
	{
		int set1 = find(subsets, edge[i].src);
		int set2 = find(subsets, edge[i].dest);

		if (set1 == set2)
			continue;
		else
		{
		if (cheapest[set1] == -1 ||
			edge[cheapest[set1]].weight > edge[i].weight)
			cheapest[set1] = i;

		if (cheapest[set2] == -1 ||
			edge[cheapest[set2]].weight > edge[i].weight)
			cheapest[set2] = i;
		}
	}

	for (int i=0; i<V; i++)
	{
		if (cheapest[i] != -1)
		{
			int set1 = find(subsets, edge[cheapest[i]].src);
			int set2 = find(subsets, edge[cheapest[i]].dest);

			if (set1 == set2)
				continue;
			MSTweight += edge[cheapest[i]].weight;
			cout<<"Edge "<<edge[cheapest[i]].src<<" -
			"<<edge[cheapest[i]].dest<< " included in MST\n";

			Union(subsets, set1, set2);
			numTrees--;
		}
	}
}

cout<<"Weight of MST is "<<MSTweight<<endl;
return;
```

```
}

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
	Graph* graph = new Graph;
	graph->V = V;
	graph->E = E;
	graph->edge = new Edge[E];
	return graph;
}

int find(struct subset subsets[], int i)
{
	if (subsets[i].parent != i)
	subsets[i].parent =
					find(subsets, subsets[i].parent);

	return subsets[i].parent;
}

void Union(struct subset subsets[], int x, int y)
{
	int xroot = find(subsets, x);
	int yroot = find(subsets, y);

	if (subsets[xroot].rank < subsets[yroot].rank)
		subsets[xroot].parent = yroot;
	else if (subsets[xroot].rank > subsets[yroot].rank)
		subsets[yroot].parent = xroot;

	else
	{
		subsets[yroot].parent = xroot;
		subsets[xroot].rank++;
	}
}

int main()
{


   int V = 14;
	int E = 21;
	Graph* graph = createGraph(V, E);
```

```
    cout<<"Enter 'from' to and weight for edges"<<endl;
    for (int i = 0; i < E; i++)
    {
        cin>>graph->edge[i].src;
        cin>>graph->edge[i].dest;
        cin>>graph->edge[i].weight;
    }
        cout<<"Edges included in minimum spanning tree are"<<endl;
        boruvkaMST(graph);

        return 0;
}
```

Q.8 TSP Tour

Write a program to verify that a candidate solution for a TSP tour is correct. The input for this program will consist of a graph, with the cities as the vertices and the connecting roads as edges, and the certificate. The graph may be expressed as an $N \times N$ matrix with the $(i, j)^{th}$ entry indicating the distance between the cities numbered *i* and *j*. If no road directly connects this pair of cities, the matrix entry will be 0. The certificate consists of a sequential list of the cities visited during the tour and the target value that the tour is not to exceed. Demonstrate that your program decides this question in O(P(N)) time.

```
4
Enter the graph details
Enter the edge cost of [0,1] pair:
10
Enter the edge cost of [0,2] pair:
15
Enter the edge cost of [0,3] pair:
5
Enter the edge cost of [1,2] pair:
10
Enter the edge cost of [1,3] pair:
20
Enter the edge cost of [2,3] pair:
10
Enter the total maximum cost:
80
Enter the path:
Enter '-1' when all nodes of the path are entered.
0
3
2
1
0
-1
Accepted!
```

Fig 8.1: Output of TSP program on user input

```
Enter the graph details
Enter the edge cost of [0,1] pair:
30
Enter the edge cost of [0,2] pair:
20
Enter the edge cost of [0,3] pair:
15
Enter the edge cost of [1,2] pair:
20
Enter the edge cost of [1,3] pair:
25
Enter the edge cost of [2,3] pair:
10
Enter the total maximum cost:
75
Enter the path:
Enter '-1' when all nodes of the path are entered.
3
1
2
0
3
-1
Total cost of the path exceeds the maximumcost allowed.
Rejected!
```

Fig 8.2: Output of TSP program on user input

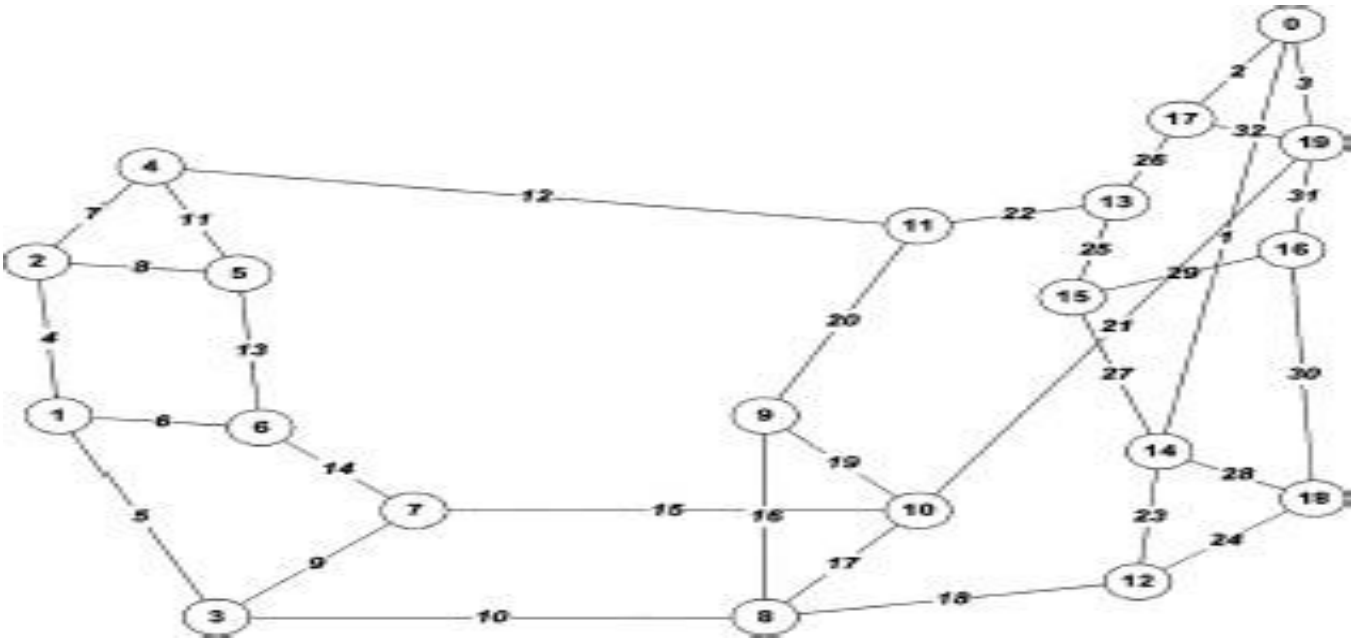
Code:

```
#include<iostream>
using namespace std;

int main(){
    int max, matSize, t=0;

    cout<<"Enter the number of nodes: "<<endl;
    cin>>matSize;

    int g[matSize][matSize], s[matSize], path[matSize];
    cout<<"Enter the graph details"<<endl;

    for(int i=0; i<matSize; i++){
        for (int j=i; j<matSize; j++){
            if(i==j){
                g[i][j] = 0;
            }
            else{
               cout<<"Enter the edge cost of ["<<i<<","<<j<<"] pair:";
                cin>>g[i][j];
                g[j][i] = g[i][j];
            }
        }
    }

    cout<<"Enter the total maximum cost: "<<endl;
    cin>>max;

    cout<<"Enter the path: "<<endl;
    cout<<"Enter '-1' when all nodes of the path are entered."<<endl;

    int l; // length of the path
    int visited[matSize];
    for(int i=0; i<matSize; i++)
        visited[i] = 0;
```

```
    for(l=0;;l++){
        int x;
        cin>>x;
        if(x==-1)
            break;
        path[l] = x;
        visited[x]=1;
    }

    if((l!= matSize+1) || (path[l-1]!=path[0])){
        cout<<" Not a complete cycle."<<endl;
        cout<<"Rejected "<<endl;
        return 0;
    }
    for(int i=0; i<matSize; i++){
        if(visited[i] == 0){
            cout<<"All nodes are not visited."<<endl;
            cout<<"Rejected"<<endl;
            return 0;
        }
    }

    for(int i=0; i<matSize; i++){
        t = t + g[path[i]][path[i+1]];
        if(t > max){
            cout<<"Total cost of the path exceeds the maximumcost
                allowed."<<endl;
            cout<<"Rejected!"<<endl;
            break;
        }
    }
    if (t <= max){
        cout<<"Accepted!"<<endl;
    }
    return 0;
}
```

Q.9 Pathing Algorithm

Use Floyd–Warshall algorithm (also known as Floyd's algorithm) to compute all pair **shortest path** for all these following standard network



ARPA Network

The following matrix shows the shortest distances between every pair of vertices

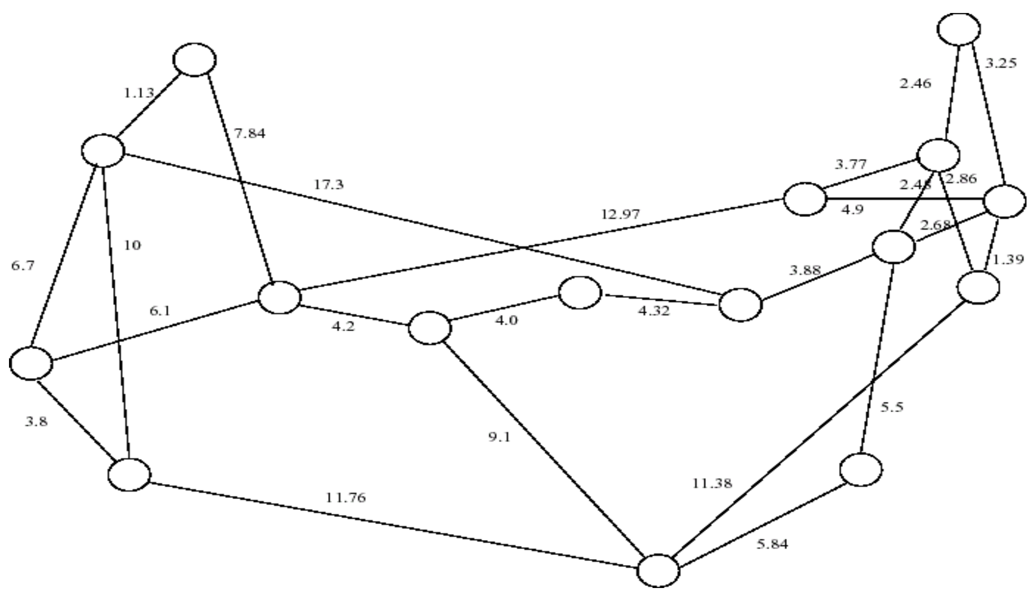
0	53	57	48	62	65	53	39	41	43	24	50	24	28	1	28	34	2	29	3
53	0	4	5	11	12	6	14	15	31	29	23	33	57	54	81	81	55	82	50
57	4	0	9	7	8	10	18	19	35	33	19	37	61	58	85	85	59	86	54
48	5	9	0	16	17	11	9	10	26	24	28	28	52	49	76	76	50	77	45
58	11	7	16	0	11	17	25	26	32	40	12	34	58	57	83	92	60	85	61
65	12	8	17	11	0	13	26	27	43	41	23	45	69	66	93	93	67	94	62
53	6	10	11	17	13	0	14	21	37	29	29	39	63	54	81	81	55	82	50
39	14	18	9	25	26	14	0	19	34	15	37	37	61	40	67	67	41	68	36
41	46	50	41	48	58	46	32	0	16	17	36	18	42	41	67	69	43	69	38
43	43	39	43	32	43	48	34	18	0	19	20	36	60	44	71	71	45	72	40
24	29	33	24	40	41	29	15	17	19	0	39	35	52	25	52	52	26	53	21
46	23	19	28	12	23	29	37	38	20	39	0	22	46	45	71	80	48	73	49
24	64	65	59	58	69	64	50	18	34	35	46	0	24	23	49	58	26	51	27
28	45	41	50	34	45	51	59	60	42	52	22	44	0	29	25	54	26	57	31
1	54	58	49	63	66	54	40	41	44	25	51	23	29	0	27	35	3	28	4
28	70	66	75	59	70	76	67	68	67	52	47	50	25	27	0	29	30	55	31
34	81	85	76	88	93	81	67	69	71	52	76	54	54	35	29	0	36	30	31
2	55	59	50	60	67	55	41	43	45	26	48	26	26	3	30	36	0	31	5
29	82	86	77	82	93	82	68	42	58	53	70	24	48	28	55	30	31	0	32
3	50	54	45	61	62	50	36	38	40	21	53	27	31	4	31	31	5	32	0

Fig 9.1: All pair shortest path for above ARPA Network

Code:

```
function [dist] = floydWarshall(graph)
    [~,V]=size(graph);
    dist=zeros(20,20);
    for i = 1:V
        for j = 1:V
            dist(i,j) = graph(i,j);
        end
    end
    for k = 1:V
        for i = 1:V
                for j = 1:V
                     if dist(i,j) > ( dist(i,k) + dist(k,j) )
                       dist(i,j) = dist(i,k) + dist(k,j);
                    end
            end
        end
    end
end
INF=999;
graph1 = [
0,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,1,INF,INF,2,INF,3;
INF,0,4,5,INF,INF,6,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF;
INF,4,0,INF,7,8,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF;
INF,5,INF,0,INF,INF,INF,9,10,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF;
INF,INF,7,INF,0,11,INF,INF,INF,INF,INF,12,INF,INF,INF,INF,INF,INF,INF,INF;
INF,INF,8,INF,11,0,13,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF;
INF,6,INF,INF,INF,13,0,14,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF;
INF,INF,INF,9,INF,INF,14,0,INF,INF,15,INF,INF,INF,INF,INF,INF,INF,INF,INF;
INF,INF,INF,10,INF,INF,INF,INF,0,16,17,INF,18,INF,INF,INF,INF,INF,INF,INF;
INF,INF,INF,INF,INF,INF,INF,INF,16,0,19,20,INF,INF,INF,INF,INF,INF,INF,INF;
INF,INF,INF,INF,INF,INF,INF,INF,17,19,0,INF,INF,INF,INF,INF,INF,INF,INF,21;
INF,INF,INF,INF,12,INF,INF,INF,INF,20,INF,0,INF,22,INF,INF,INF,INF,INF,INF;
INF,INF,INF,INF,INF,INF,INF,INF,18,INF,INF,INF,0,INF,23,INF,INF,INF,24,INF;
INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,22,INF,0,INF,25,INF,26,INF,INF;
1,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,23,INF,0,27,INF,INF,28,INF;
INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,25,27,0,29,INF,INF,INF;
INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,29,0,INF,30,31;
2,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,26,INF,INF,INF,0,INF,32;
INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,INF,24,INF,28,INF,30,INF,0,INF;
3,INF,INF,INF,INF,INF,INF,INF,INF,INF,21,INF,INF,INF,INF,INF,31,32,INF,0 ];
x=999;
graph2= [
0,1.13,x,7.84,x,x,x,x,x,x,x,x,x,x,x,x ;
1.13,0,6.7,x,10,x,x,17.3,x,x,x,x,x,x,x,x ;
x,6.7,0,6.1,3.8,x,x,x,x,x,x,x,x,x,x,x ;
7.84,x,6.1,0,x,4.2,x,x,x,x,12.97,x,x,x,x,x ;
x,10,3.8,x,0,x,x,x,11.76,x,x,x,x,x,x,x ;
```

```
x,x,x,4.2,x,0,4.0,x,9.1,x,x,x,x,x,x,x ;
x,x,x,x,x,4.0,0,4.32,x,x,x,x,x,x,x,x ;
x,17.3,x,x,x,x,4.32,0,x,x,x,3.88,x,x,x,x ;
x,x,x,x,11.76,9.1,x,x,0,5.84,x,x,11.38,x,x,x ;
x,x,x,x,x,x,x,x,5.84,0,x,5.5,x,x,x,x ;
x,x,x,12.97,x,x,x,x,x,x,x,x,x,3.77,4.9,x ;
x,x,x,x,x,x,x,3.88,x,5.5,x,0,x,2.48,2.68,x ;
x,x,x,x,x,x,x,x,11.38,x,x,x,0,2.86,1.39,x ;
x,x,x,x,x,x,x,x,x,x,3.77,2.38,2.86,0,x,2.46 ;
x,x,x,x,x,x,x,x,x,x,4.9,2.68,1.39,x,0,3.25 ;
x,x,x,x,x,x,x,x,x,x,x,x,x,2.46,3.25,0];
sprintf("The following matrix shows the shortest distances between every pair of vertices")
nsf=floydWarshall(graph1)
arpa=floydWarshall(graph2)
```

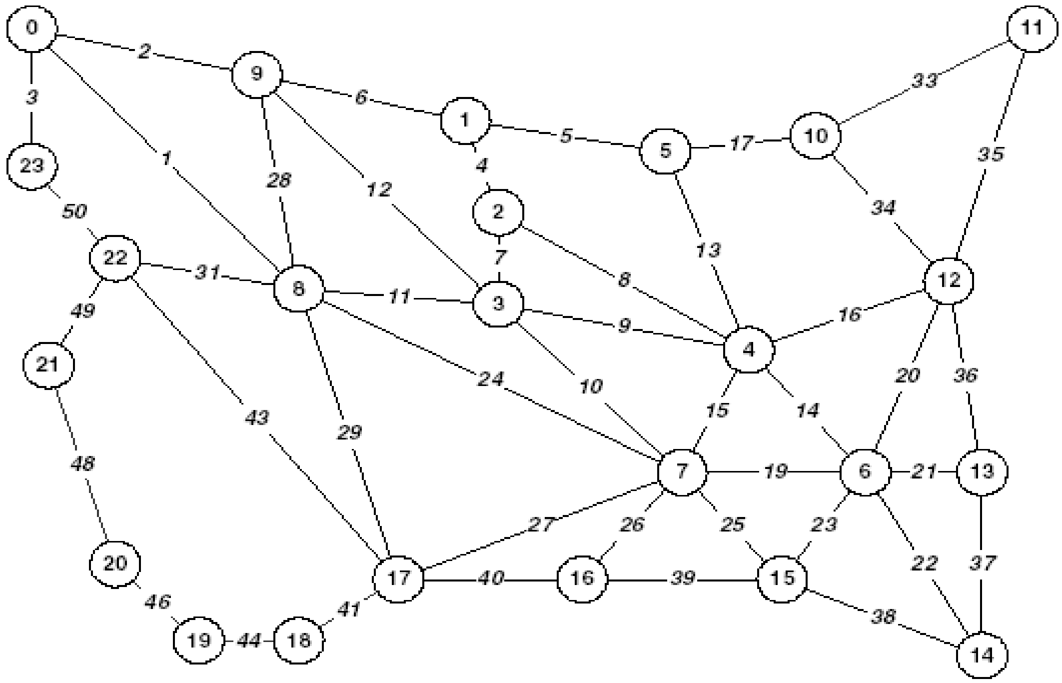


The following matrix shows the shortest distances between every pair of vertices

0	26.98	26.68	21.4	17.6	19.14	14.94	13.7	9.38	5.84	11.75	7.98	10.44	5.5	8.18	9.57
26.98	0	1.13	7.83	11.13	7.84	12.04	16.04	18.43	21.14	20.81	24.58	27.04	22.31	24.99	26.38
26.68	1.13	0	6.7	10	8.97	13.17	17.17	17.3	21.76	21.94	23.66	26.12	21.18	23.86	25.25
21.4	7.83	6.7	0	3.8	6.1	10.3	14.3	18.62	15.56	19.07	22.84	25.3	22.5	23.97	25.36
17.6	11.13	10	3.8	0	9.9	14.1	18.1	22.42	11.76	22.87	25.58	27.78	23.1	24.53	23.14
19.14	7.84	8.97	6.1	9.9	0	4.2	8.2	12.52	13.3	12.97	16.74	19.2	16.4	17.87	19.26
14.94	12.04	13.17	10.3	14.1	4.2	0	4	8.32	9.1	17.17	14.68	17.14	12.2	14.88	16.27
13.7	16.04	17.17	14.3	18.1	8.2	4	0	4.32	13.1	14.45	10.68	13.14	8.2	10.88	12.27
9.38	18.43	17.3	18.62	22.42	12.52	8.32	4.32	0	15.22	10.13	6.36	8.82	3.88	6.56	7.95
5.84	21.14	21.76	15.56	11.76	13.3	9.1	13.1	15.22	0	17.59	13.82	16.02	11.34	12.77	11.38
11.75	20.81	21.94	19.07	22.87	12.97	17.17	14.45	10.13	17.59	0	3.77	6.23	6.25	4.9	6.29
7.98	24.58	23.66	22.84	25.58	16.74	14.68	10.68	6.36	13.82	3.77	0	2.46	2.48	4.25	2.86
10.44	27.04	26.12	25.3	27.78	19.2	17.14	13.14	8.82	16.02	6.23	2.46	0	4.94	3.25	4.64
5.5	22.31	21.18	22.5	23.1	16.4	12.2	8.2	3.88	11.34	6.25	2.48	4.94	0	2.68	4.07
8.18	24.99	23.86	23.97	24.53	17.87	14.88	10.88	6.56	12.77	4.9	4.25	3.25	2.68	0	1.39
9.57	26.38	25.25	25.36	23.14	19.26	16.27	12.27	7.95	11.38	6.29	2.86	4.64	4.07	1.39	0

NSF Network

Fig 9.2: All pair shortest path for above NSF Network



NATIONAL NETWORK

The following matrix shows the shortest distances between every pair of vertices

0	11	15	15	23	16	37	25	4	5	33	66	39	58	59	50	51	33	74	118	132	84	35	3
8	0	4	11	12	5	26	21	12	6	22	55	28	47	48	46	47	41	82	126	140	92	43	11
12	4	0	7	8	9	22	17	16	10	26	59	24	43	44	42	43	44	85	129	144	96	47	15
12	11	7	0	9	16	23	10	11	17	33	60	25	44	45	35	36	37	78	122	139	91	42	15
20	12	8	9	0	13	14	15	20	18	30	51	16	35	36	37	41	42	83	127	148	100	51	23
13	5	9	16	13	0	27	26	17	11	17	50	29	48	49	50	52	46	87	131	145	97	48	16
34	26	22	23	14	27	0	19	34	32	44	55	20	21	22	23	45	46	87	131	162	114	65	37
22	21	17	10	15	26	19	0	21	27	43	66	31	40	41	25	26	27	68	112	149	101	52	25
1	12	16	11	20	17	34	21	0	6	34	67	36	55	56	46	47	29	70	114	128	80	31	4
2	6	10	12	18	11	32	22	6	0	28	61	34	53	54	47	48	35	76	120	134	86	37	5
30	22	26	33	30	17	44	43	34	28	0	33	34	65	66	67	69	63	104	148	162	114	65	33
63	55	59	60	51	50	55	66	67	61	33	0	35	71	77	78	92	93	134	178	195	147	98	66
36	28	24	25	16	29	20	31	36	34	34	35	0	36	42	43	57	58	99	143	164	116	67	39
55	47	43	44	35	48	21	40	55	53	65	71	36	0	37	44	66	67	108	152	183	135	86	58
56	48	44	45	36	49	22	41	56	54	66	72	37	43	0	38	67	68	109	153	184	136	87	59
40	44	42	35	37	49	23	25	44	38	66	78	43	44	45	0	39	52	93	137	172	124	75	43
62	54	50	51	42	55	46	57	62	60	39	61	26	62	68	69	0	40	81	125	171	132	83	65
51	50	46	39	41	54	27	29	50	56	71	82	47	48	49	50	40	0	41	85	131	92	43	54
92	91	87	80	82	95	68	70	91	97	112	123	88	89	90	91	81	41	0	44	90	133	84	95
136	135	131	124	126	139	112	114	135	141	156	167	132	133	134	135	125	85	44	0	46	94	128	139
129	140	144	139	148	145	158	149	128	134	162	195	164	179	180	174	171	131	90	46	0	48	97	132
81	92	96	91	100	97	114	101	80	86	114	147	116	135	136	126	127	92	133	94	48	0	49	84
32	43	47	42	51	48	65	52	31	37	65	98	67	86	87	77	78	43	84	128	97	49	0	35
2	8	12	12	20	13	34	22	1	2	30	63	36	55	56	47	48	30	71	115	129	81	32	0

Fig 9.3: All pair shortest path for above National Network