**CODE:**

#include<stdio.h>

int main(){

int i, j, count, temp, number[25];

printf("enter number of elements :");

scanf("%d",&count);

printf("Enter %d elements: ", count);

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

scanf("%d",&number[i]);

for(i=1;i<count;i++){

temp=number[i];

j=i-1;

while((temp<number[j])&&(j>=0)){

number[j+1]=number[j];

j=j-1;

}

number[j+1]=temp;

}

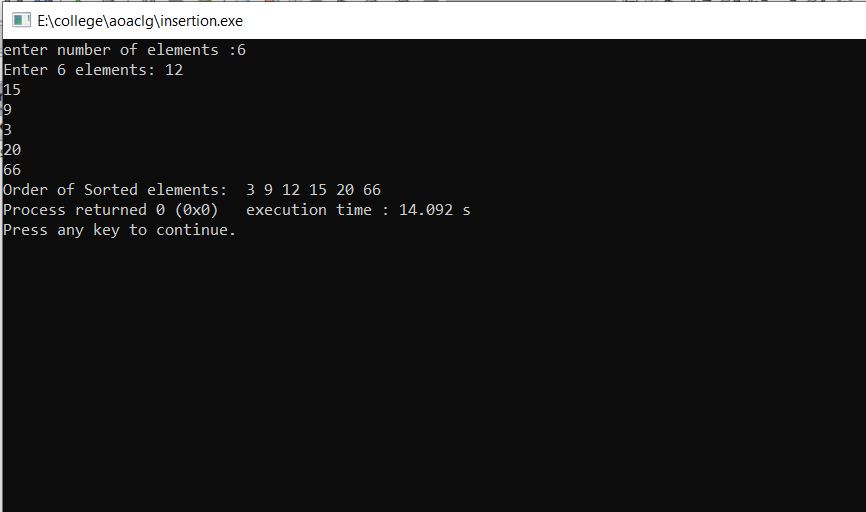
printf("Order of Sorted elements: ");

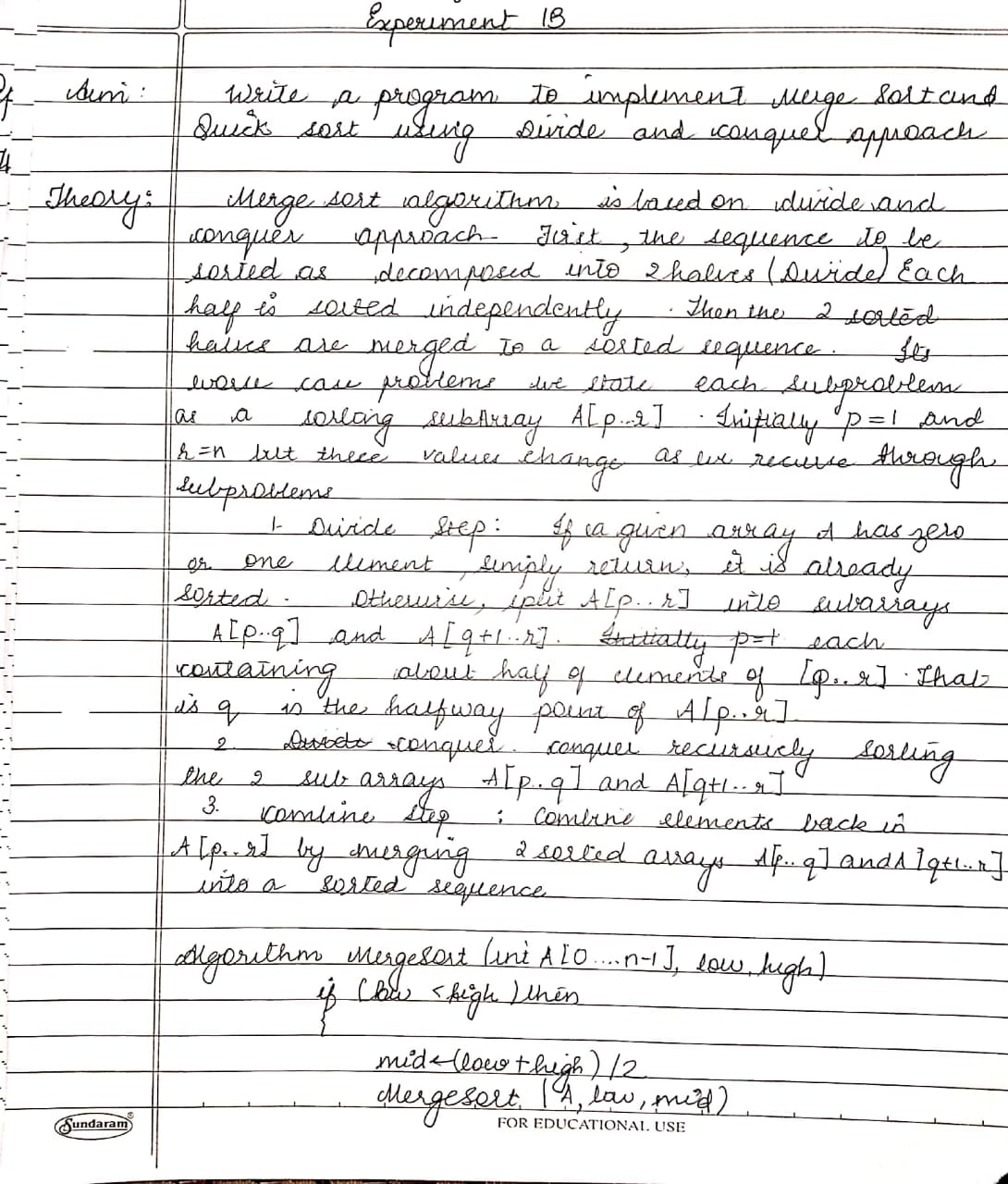
for(i=0;i<count;i++)

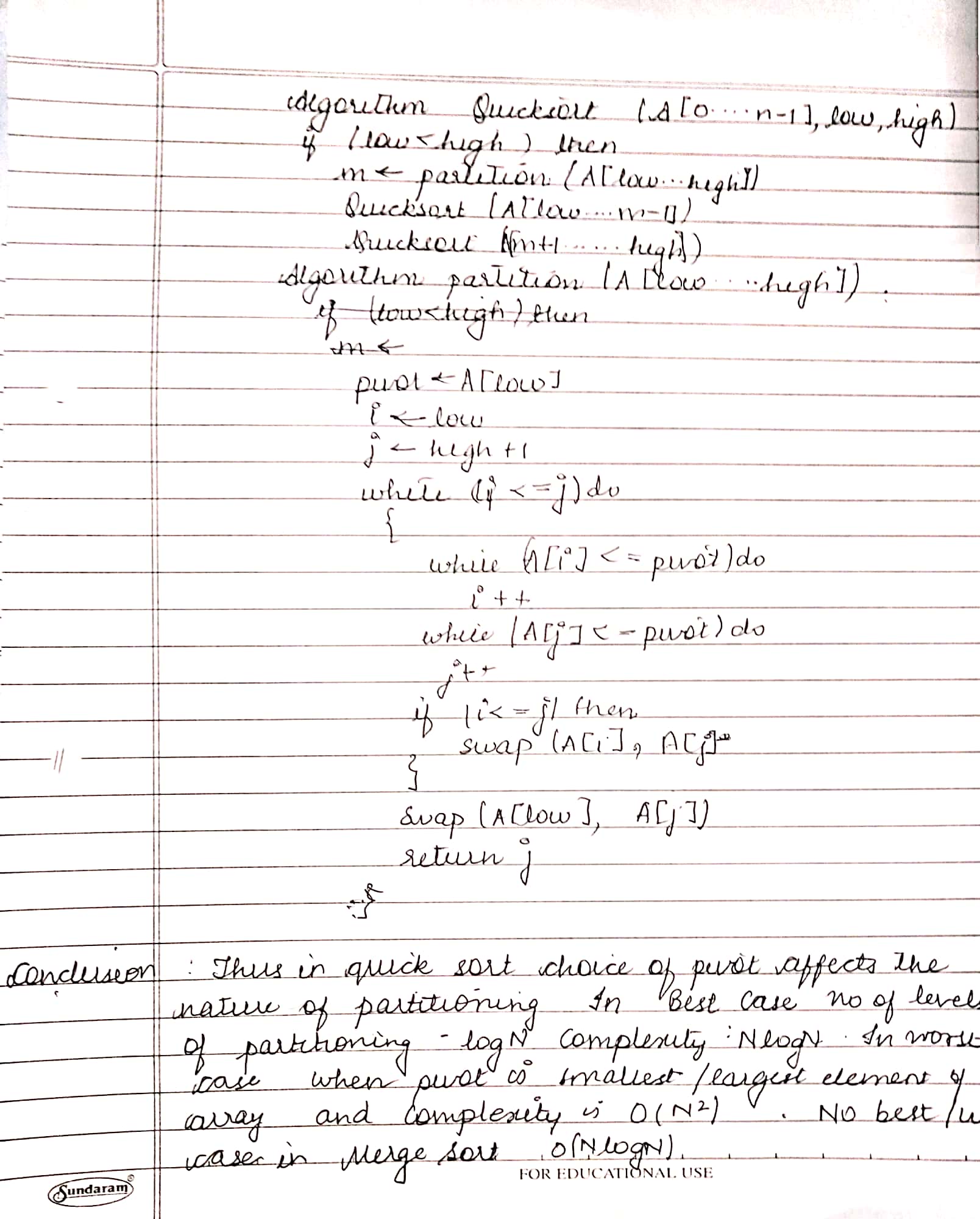
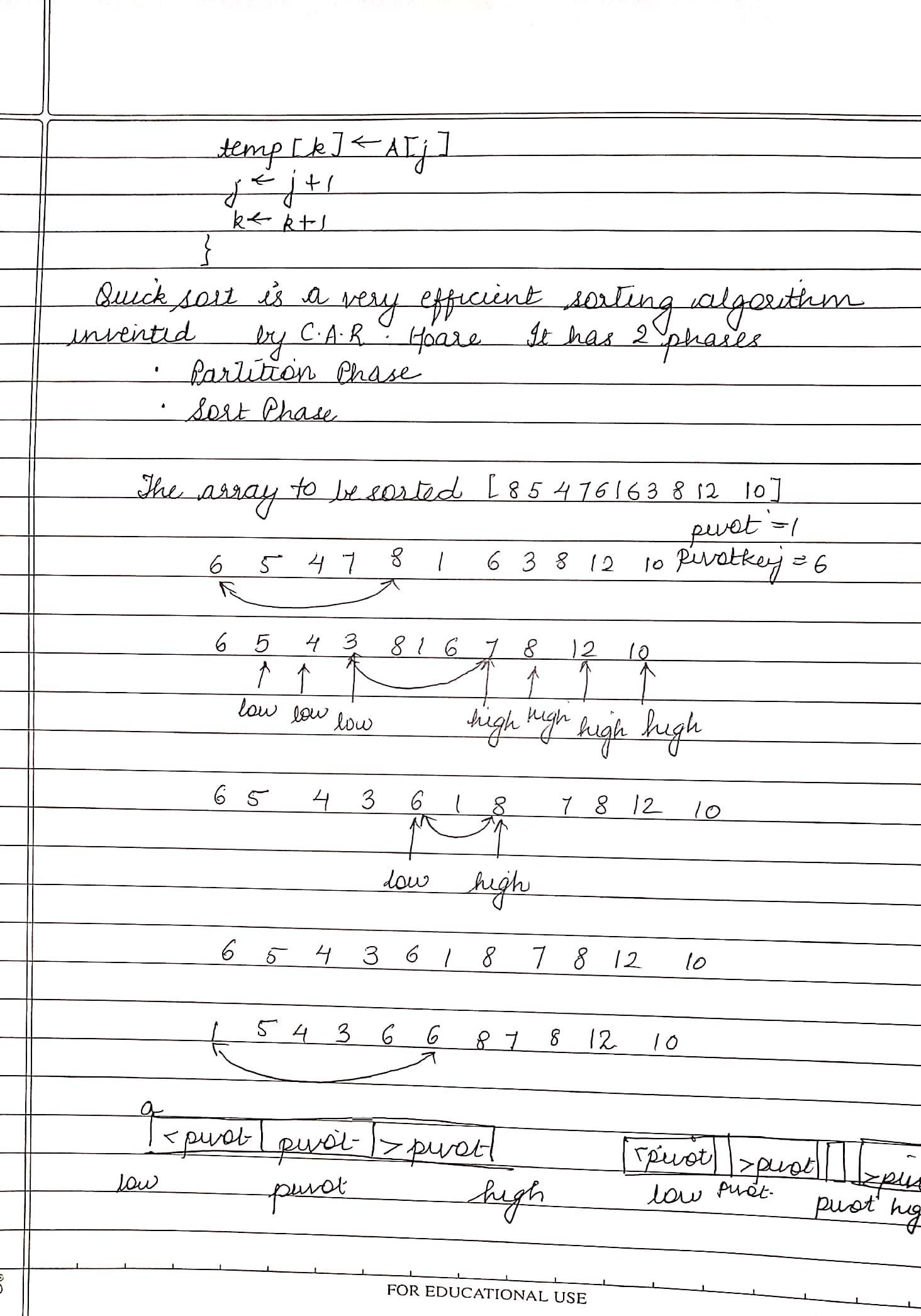
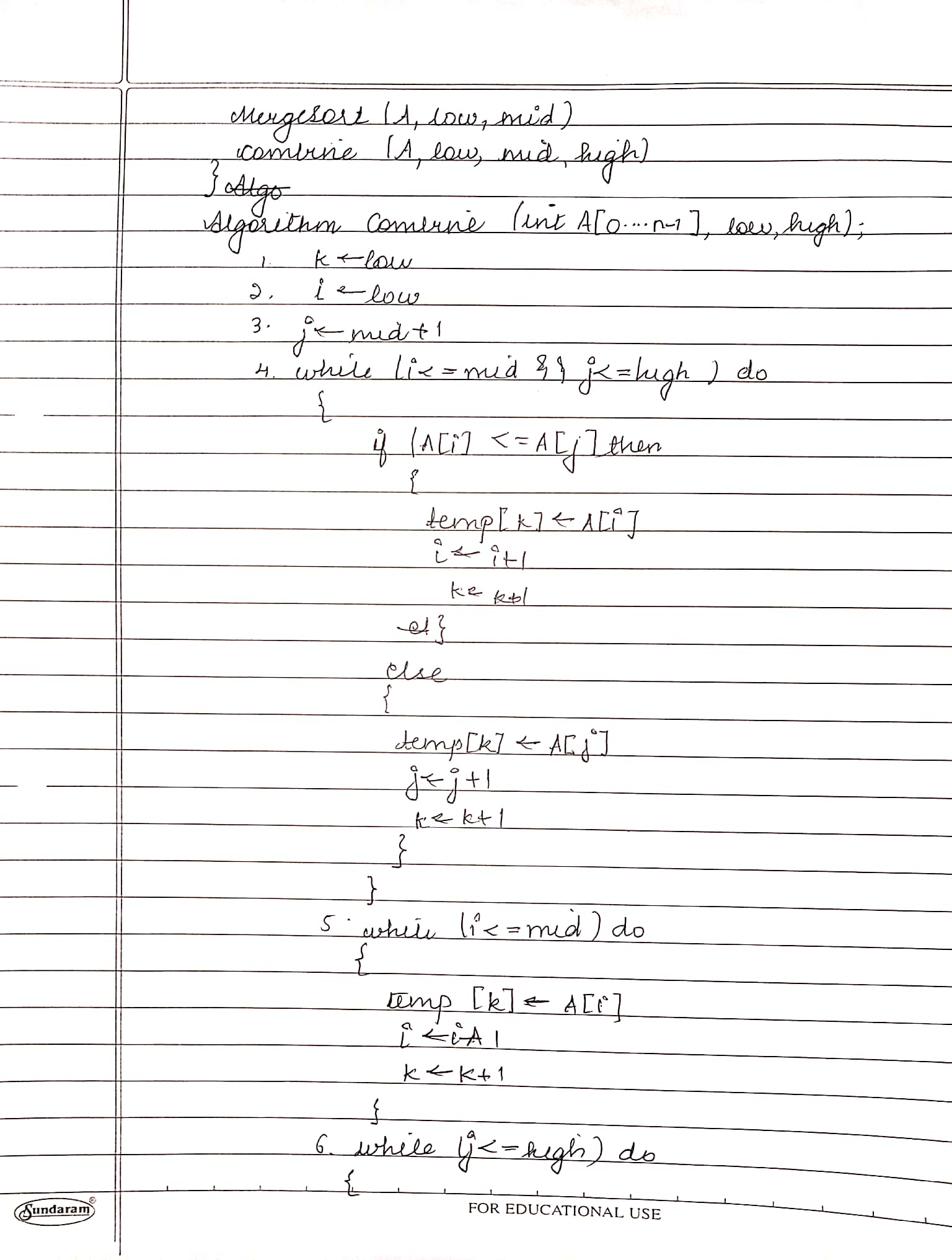
printf(" %d",number[i]);

return 0;

}

****



**CODE:**

#include<stdio.h>

int a[50];

int size;

void merge(int beg,int mid,int end)

{

int i=beg,j=mid+1,index=beg,temp[size],k;

while(i<=mid && j<=end){

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

temp[index]=a[i];

i++;

}

else{

temp[index]=a[j];

j++;

}

index++;

}

//if(i>=mid){

while(j<=end){

temp[index]=a[j];

j++;

index++;

}

//else{

while(i<=mid){

temp[index]=a[i];

i++;

index++;

}

for(k=beg;k<index;k++){

a[k]=temp[k];

}

}

void merge\_sort(int beg,int end){

int mid;

if(beg<end){

mid=(beg+end)/2;

merge\_sort(beg,mid);

merge\_sort(mid+1,end);

merge(beg,mid,end); }

}

void main()

{

int n,i;

printf("\n enter the size of array : ");

scanf("%d",&n);

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

scanf("%d",&a[i]);

}

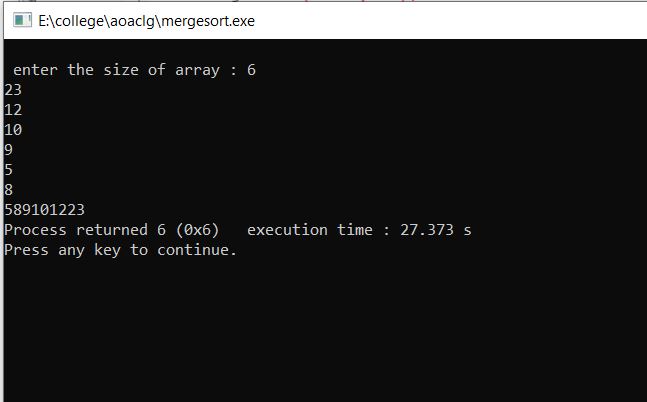
merge\_sort(0,n-1);

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

printf("%d",a[i]);

}

}



QUICK SORT

#include<stdio.h>

void quicksort(int number[25],int first,int last){

int i, j, pivot, temp;

if(first<last){

pivot=first;

i=first;

j=last;

while(i<j){

while(number[i]<=number[pivot]&&i<last)

i++;

while(number[j]>number[pivot])

j--;

if(i<j){

temp=number[i];

number[i]=number[j];

number[j]=temp;

}

}

temp=number[pivot];

number[pivot]=number[j];

number[j]=temp;

quicksort(number,first,j-1);

quicksort(number,j+1,last);

}

}

int main(){

int i, count, number[25];

printf("Enter the number of elements: ");

scanf("%d",&count);

printf("Enter %d elements: ", count);

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

scanf("%d",&number[i]);

quicksort(number,0,count-1);

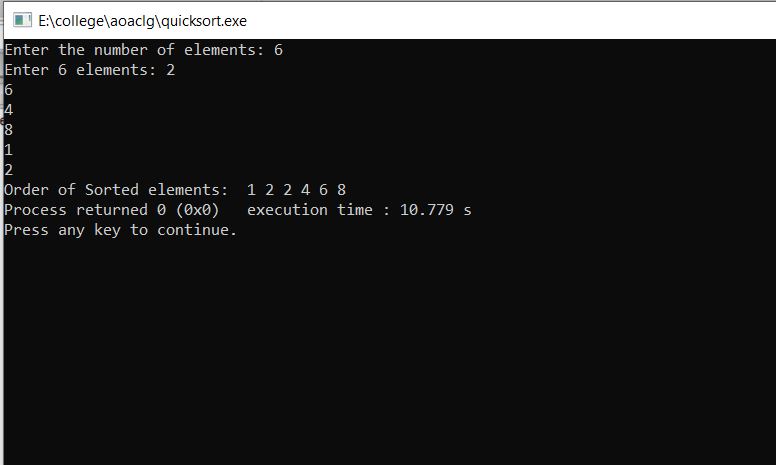
printf("Order of Sorted elements: ");

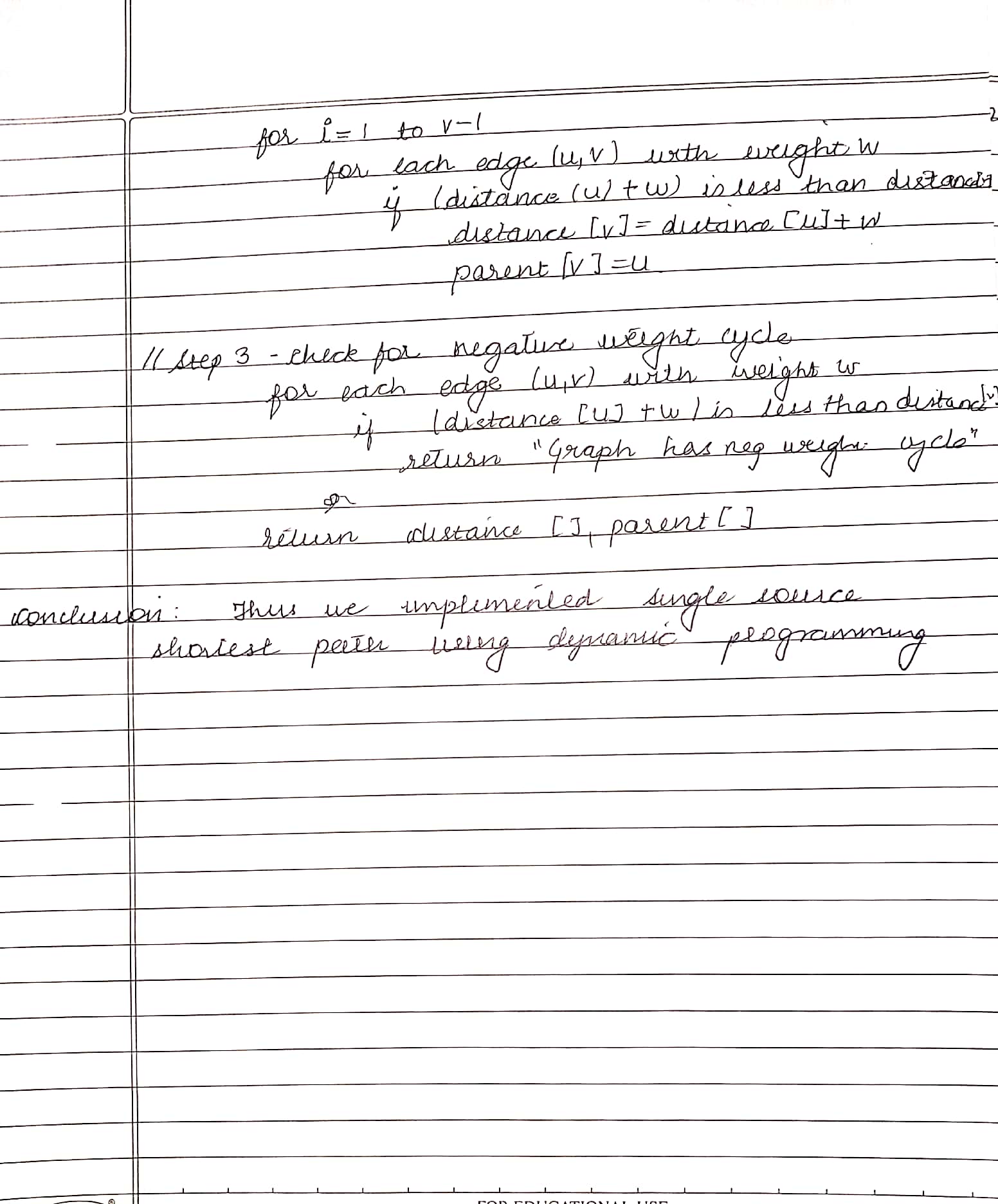
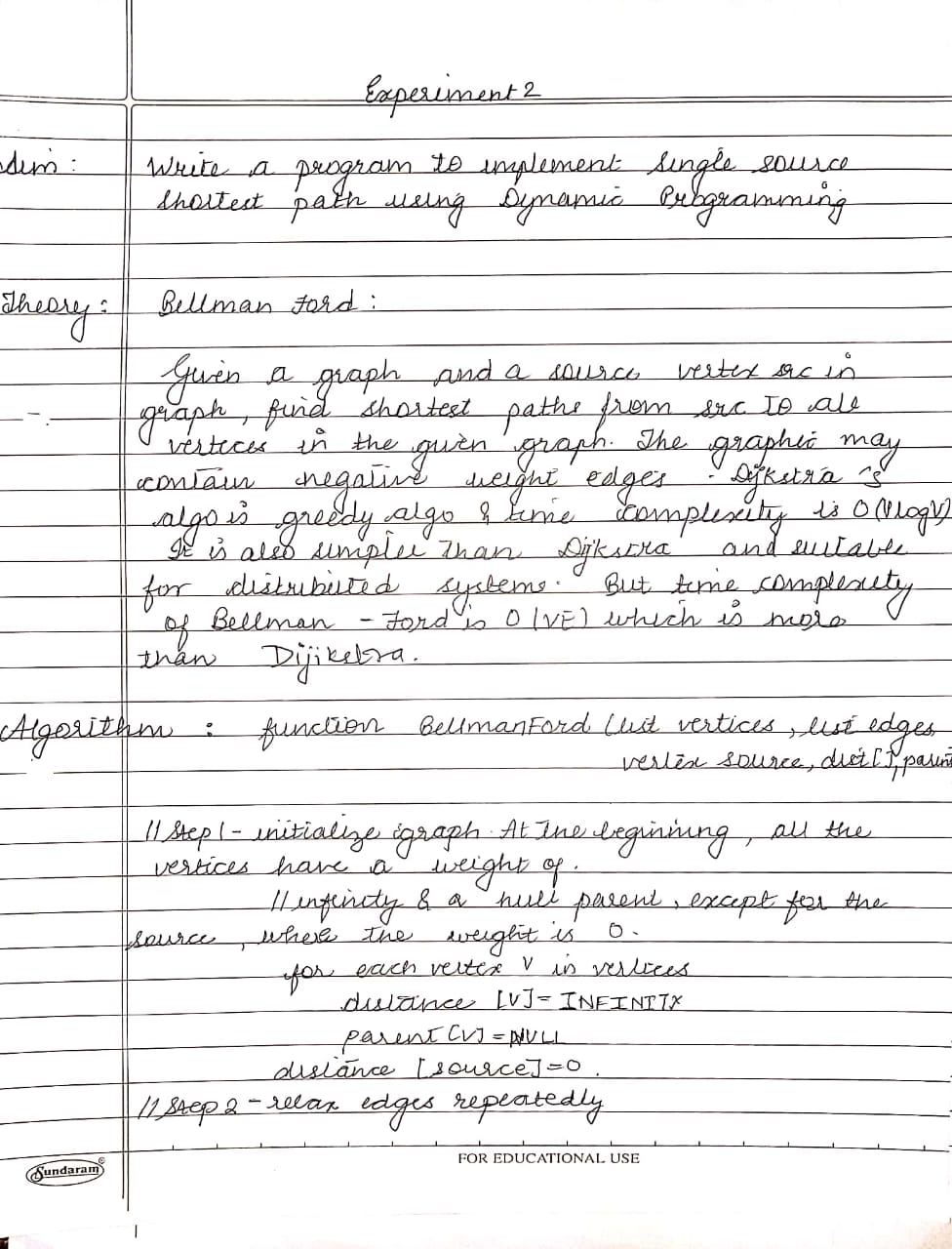
for(i=0;i<count;i++)

printf(" %d",number[i]);

return 0;

}





**CODE:**

#include <stdio.h>

#include <stdlib.h>

int Bellman\_Ford(int G[20][20] , int V, int E, int edge[20][2])

{

int i,u,v,k,distance[20],parent[20],S,flag=1;

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

distance[i] = 1000 , parent[i] = -1 ;

printf("Enter source: ");

scanf("%d",&S);

distance[S-1]=0 ;

for(i=0;i<V-1;i++)

{

for(k=0;k<E;k++)

{

u = edge[k][0] , v = edge[k][1] ;

if(distance[u]+G[u][v] < distance[v])

distance[v] = distance[u] + G[u][v] , parent[v]=u ;

}

}

for(k=0;k<E;k++)

{

u = edge[k][0] , v = edge[k][1] ;

if(distance[u]+G[u][v] < distance[v])

flag = 0 ;

}

if(flag)

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

printf("Vertex %d -> cost = %d parent = %d\n",i+1,distance[i],parent[i]+1);

return flag;

}

int main()

{

int V,edge[20][2],G[20][20],i,j,k=0;

printf("BELLMAN FORD\n");

printf("Enter no. of vertices: ");

scanf("%d",&V);

printf("Enter graph in matrix form:\n");

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

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

{

scanf("%d",&G[i][j]);

if(G[i][j]!=0)

edge[k][0]=i,edge[k++][1]=j;

}

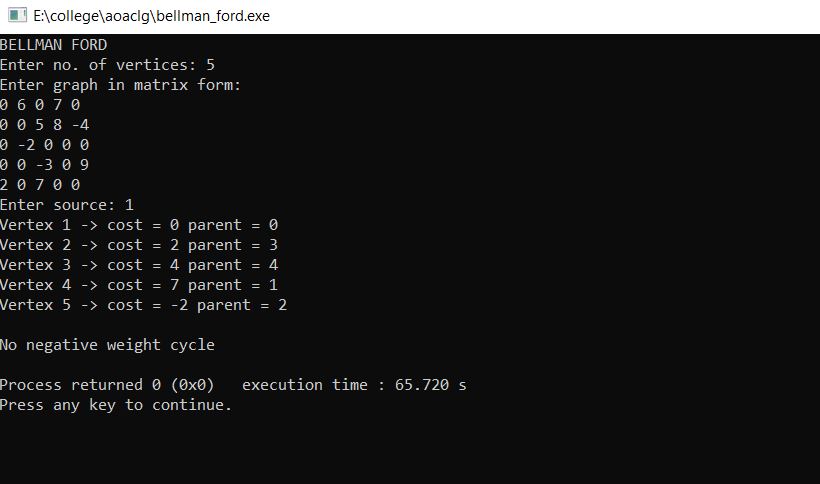
if(Bellman\_Ford(G,V,k,edge))

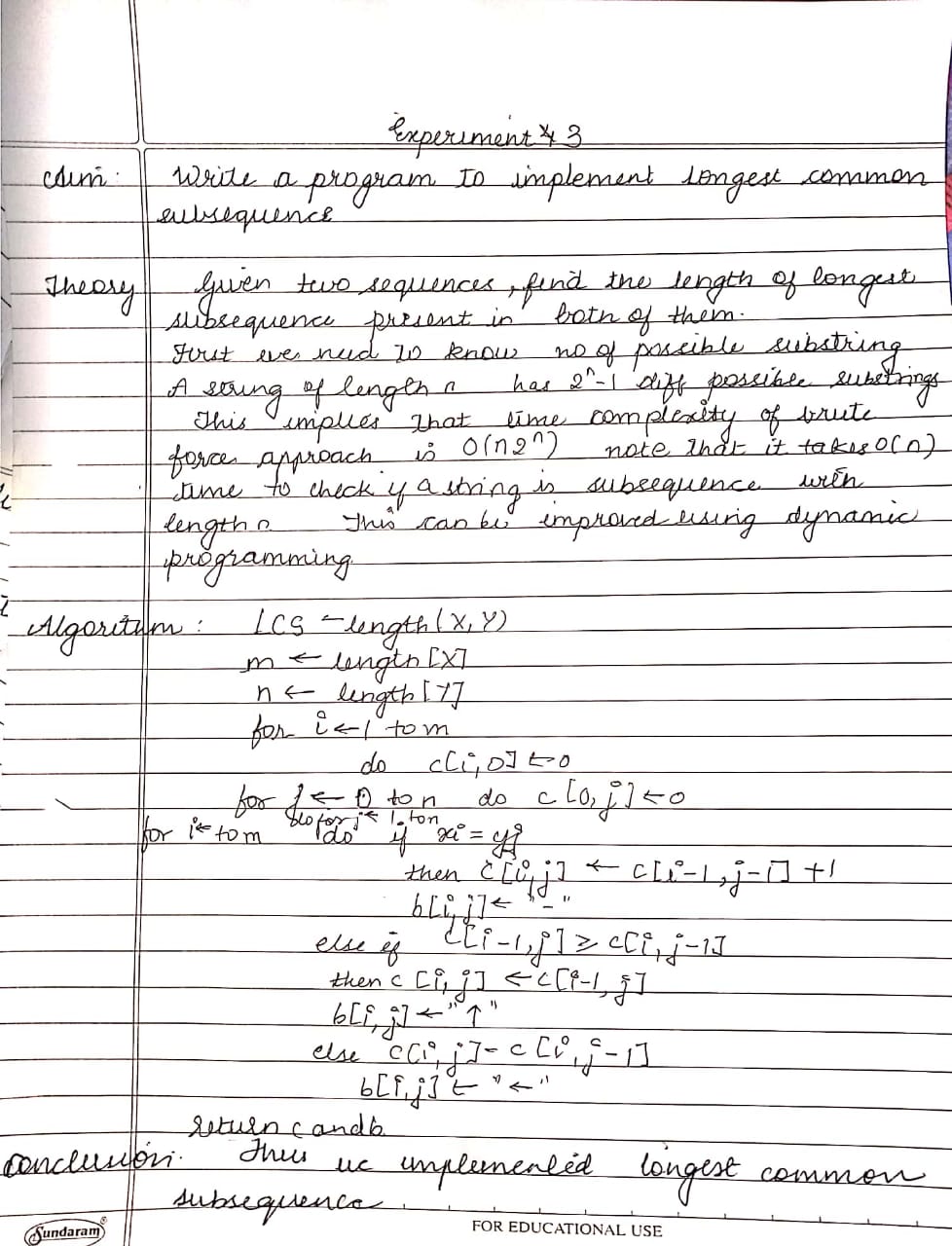
printf("\nNo negative weight cycle\n");

else printf("\nNegative weight cycle exists\n");

return 0;

}





**CODE:**

#include <stdio.h>

#include <string.h>

int i, j, m, n, LCS\_table[20][20];

char S1[20] = "ACADB", S2[20] = "CBDA", b[20][20];

void lcs()

{

m = strlen(S1);

n = strlen(S2);

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

LCS\_table[i][0] = 0;

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

LCS\_table[0][i] = 0;

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

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

{

if (S1[i - 1] == S2[j - 1])

{

LCS\_table[i][j] = LCS\_table[i - 1][j - 1] + 1;

}

else if (LCS\_table[i - 1][j] >= LCS\_table[i][j - 1])

{

LCS\_table[i][j] = LCS\_table[i - 1][j];

}

else

{

LCS\_table[i][j] = LCS\_table[i][j - 1];

}

}

int index = LCS\_table[m][n];

char lcs[index + 1];

lcs[index] = '\0';

int i = m, j = n;

while (i > 0 && j > 0)

{

if (S1[i - 1] == S2[j - 1])

{

lcs[index - 1] = S1[i - 1];

i--;

j--;

index--;

}

else if (LCS\_table[i - 1][j] > LCS\_table[i][j - 1])

i--;

else

j--;

}

printf("S1 : %s \nS2 : %s \n", S1, S2);

printf("LCS: %s", lcs);

}

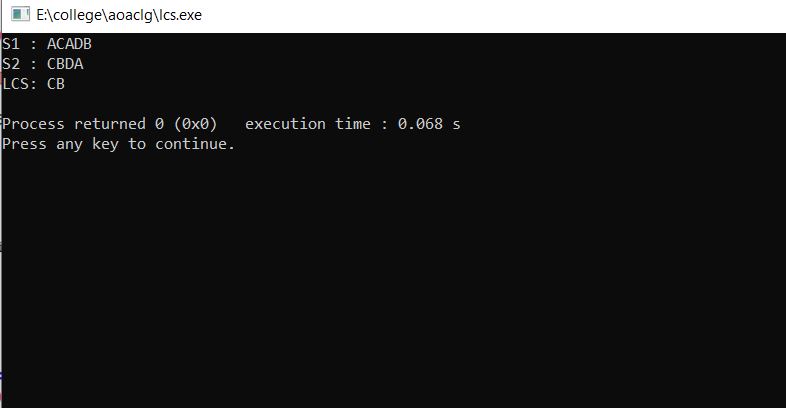
int main()

{

lcs();

printf("\n");

}



**EXPERIMENT 4**

**Aim: Write a program to implement Minimum cost spanning trees-Kruskal and prim’s algorithm.**

**Theory:**

**Graphs :**A graph is a set of *vertices* and *edges* which connect them. We write:

**G = (V,E)** where **V** is the set of vertices and the set of edges,

**E = { (vi, vj) }** where **vi** and **vj** are in **V**.

**Paths :**A *path*, p, of length, k, through a graph is a sequence of connected vertices:

**p = <v0,v1,...,vk>** where, for all **i** in (0,**k**-1):

**(vi,vi+1)** is in **E**.

**Cycles :** A graph contains no *cycles* if there is no path of non-zero length through the graph, p = <v0,v1,...,vk> such that v0 = vk.

**Spanning Trees :** A *spanning tree* of a graph, G, is a set of |V|-1 edges that connect all vertices of the graph.

**Minimum Spanning Tree :**

Given a connected, undirected graph, a spanning tree of that graph is a subgraph which is a tree and connects all the vertices together. A single graph can have many different spanning trees. We can also assign a *weight* to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A minimum spanning tree (MST) or minimum weight spanning tree is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a minimum spanning forest, which is a union of minimum spanning trees for its connected components. In general, it is possible to construct multiple spanning trees for a graph, **G**. If a cost, **cij**, isassociated with each edge, **eij= (vi,vj)**, then the minimum spanning tree is the set of edges, **Espan**,forming a spanning tree, such that: **C = sum( cij**| all **eij**in **Espan**) is a minimum.

**KRUSKAL'S ALGORITHM:**

**It** is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm.

**ALGORITHM:**

KRUSKAL-MST(V,E,w)

Initialization and setup

T ← NULL

for each vertex v ∈ V do

MAKE-SET(v)

Sort the edges in E into non-decreasing order of weight

for each edge (u,v) ∈ E in non-decreasing order of weight do

if FIND-SET(u) 6= FIND-SET(v) then

T ← T ∪{(u,v)}

UNION(u,v)

return T

**CODE FOR KRUSKAL’S :**

#include<stdio.h>

#define MAX 30

typedef struct edge

{

int u,v,w;

}edge;

typedef struct edgelist

{

edge data[MAX];

int n;

}edgelist;

edgelist elist;

int G[MAX][MAX],n;

edgelist spanlist;

void kruskal();

int find(int belongs[],int vertexno);

void union1(int belongs[],int c1,int c2);

void sort();

void print();

void main()

{

int i,j,total\_cost;

printf("\nEnter number of vertices:");

scanf("%d",&n);

printf("\nEnter the adjacency matrix:\n");

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

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

scanf("%d",&G[i][j]);

kruskal();

print();

}

void kruskal()

{

int belongs[MAX],i,j,cno1,cno2;

elist.n=0;

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

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

{

if(G[i][j]!=0)

{

elist.data[elist.n].u=i;

elist.data[elist.n].v=j;

elist.data[elist.n].w=G[i][j];

elist.n++;

}

}

sort();

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

belongs[i]=i;

spanlist.n=0;

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

{

cno1=find(belongs,elist.data[i].u);

cno2=find(belongs,elist.data[i].v);

if(cno1!=cno2)

{

spanlist.data[spanlist.n]=elist.data[i];

spanlist.n=spanlist.n+1;

union1(belongs,cno1,cno2);

}

}

}

int find(int belongs[],int vertexno)

{

return(belongs[vertexno]);

}

void union1(int belongs[],int c1,int c2)

{

int i;

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

if(belongs[i]==c2)

belongs[i]=c1;

}

void sort()

{

int i,j;

edge temp;

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

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

if(elist.data[j].w>elist.data[j+1].w)

{

temp=elist.data[j];

elist.data[j]=elist.data[j+1];

elist.data[j+1]=temp;

}

}

void print()

{

int i,cost=0;

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

{

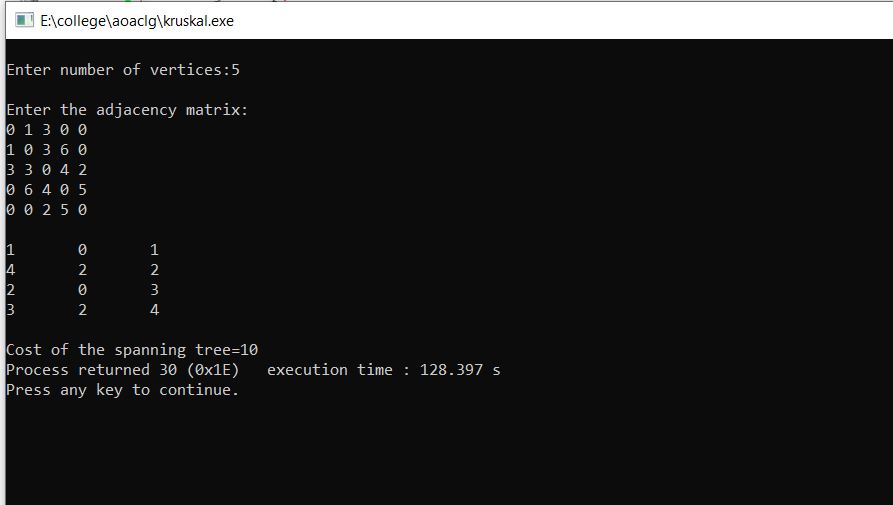
printf("\n%d\t%d\t%d",spanlist.data[i].u,spanlist.data[i].v,spanlist.data[i].w);

cost=cost+spanlist.data[i].w;

}

printf("\n\nCost of the spanning tree=%d",cost);

}



**PRIM’S ALGORITHM:**

**It** is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree thatincludes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm continuously increases the size of a tree starting with a single vertex until it spans all the vertices.

Input: A connected weighted graph with vertices V and edges E.

Initialize: **Vnew= {x},** where x is an arbitrary node (starting point) from V, **Enew= {}**

Repeat until **Vnew= V:**

oChoose edge **(u,v)** from E with minimal weight such that u is in **Vnew**and v is not

(if there are multiple edges with the same weight, choose arbitrarily)

Add v to **Vnew**, add (u, v) to **Enew**

Output: **Vnew**and **Enew**describe a minimal spanning tree

**ALGORITHM :**

MST-PRIM(G, w, r)

1. for each u V [G]

2. do key[u] ← ∞

3. π[u] ← NIL

4. key[r] ← 0

5. Q ← V [G]

6. while Q ≠ Ø

7. do u ← EXTRACT-MIN(Q)

8. for each v Adj[u]

9. do if v Q and w(u, v) < key[v]

10. then π[v] ← u

11. key[v] ← w(u, v)

**Conclusion:** The time required by Prim's algorithm is O(|V|2). It will be reduced to (|E|log|V|) if heap is used to keep {v: L(v) < infinity}.A simple implementation using an adjacency matrix graph representation and searching an array of weights to find the minimum weight edge to add requires O(*V2*) running time. Using a simplebinary heap data structure and an adjacency list representation, Prim's algorithm can be shown torun in time O(*E* log *V*) where E is the number of edges and V is the number.The time required by Kruskal's algorithm is O(|E|log|EV|). Using a simple binaryheap data structure and an adjacency list representation, Kruskal‟s algorithm can be shown to runin time O(*E* log *V*) where E is the number of edges and V is the number of vertices.

**CODE:**

#include<stdio.h>

#include<stdlib.h>

#define infinity 9999

#define MAX 20

int G[MAX][MAX],spanning[MAX][MAX],n;

int prims();

int main()

{

int i,j,total\_cost;

printf("Enter no. of vertices:");

scanf("%d",&n);

printf("\nEnter the adjacency matrix:\n");

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

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

scanf("%d",&G[i][j]);

total\_cost=prims();

printf("\nspanning tree matrix:\n");

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

{

printf("\n");

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

printf("%d\t",spanning[i][j]);

}

printf("\n\nTotal cost of spanning tree=%d",total\_cost);

return 0;

}

int prims()

{

int cost[MAX][MAX];

int u,v,min\_distance,distance[MAX],from[MAX];

int visited[MAX],no\_of\_edges,i,min\_cost,j;

//create cost[][] matrix,spanning[][]

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

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

{

if(G[i][j]==0)

cost[i][j]=infinity;

else

cost[i][j]=G[i][j];

spanning[i][j]=0;

}

//initialise visited[],distance[] and from[]

distance[0]=0;

visited[0]=1;

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

{

distance[i]=cost[0][i];

from[i]=0;

visited[i]=0;

}

min\_cost=0; //cost of spanning tree

no\_of\_edges=n-1; //no. of edges to be added

while(no\_of\_edges>0)

{

//find the vertex at minimum distance from the tree

min\_distance=infinity;

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

if(visited[i]==0&&distance[i]<min\_distance)

{

v=i;

min\_distance=distance[i];

}

u=from[v];

//insert the edge in spanning tree

spanning[u][v]=distance[v];

spanning[v][u]=distance[v];

no\_of\_edges--;

visited[v]=1;

//updated the distance[] array

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

if(visited[i]==0&&cost[i][v]<distance[i])

{

distance[i]=cost[i][v];

from[i]=v;

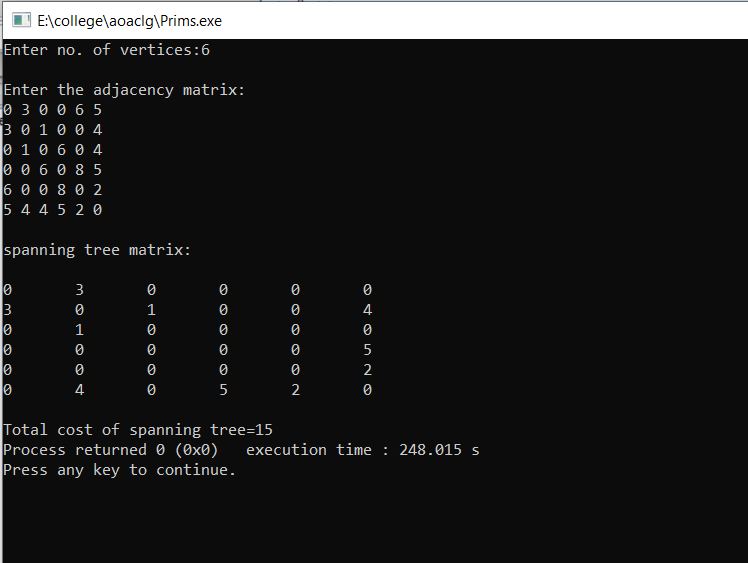
}

min\_cost=min\_cost+cost[u][v];

}

return(min\_cost);

}

****

**EXPERIMENT 5**

**Aim: Write a program to implement single source shortest path using Greedy Approach**

**THEORY:**

**DJIKSTRA ALGORITHM:**

Given a graph and a source vertex in the graph, find shortest paths from source to all vertices in the given graph.

Dijkstra’s algorithm is very similar to [Prim’s algorithm for minimum spanning tree](https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/). Like Prim’s MST, we generate a*SPT (shortest path tree)* with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

**ALGORITHM:**

1. Mark all nodes unvisited. Create a set of all the unvisited nodes called the *unvisited set*.
2. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes. Set the initial node as current.[[14]](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#cite_note-14)
3. For the current node, consider all of its unvisited neighbours and calculate their *tentative* distances through the current node. Compare the newly calculated *tentative* distance to the current assigned value and assign the smaller one. For example, if the current node *A* is marked with a distance of 6, and the edge connecting it with a neighbour *B* has length 2, then the distance to *B* through *A* will be 6 + 2 = 8. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, the current value will be kept.
4. When we are done considering all of the unvisited neighbours of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the *unvisited set* is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

**CODE:**

#include<stdio.h>

int a[9][9];

int set[9]={-1};

struct node

{

int key;

int p;

int flag;

};

int min(struct node n[9])

{

int m,min,p;

min = 999;

for (m=0; m<9;m++)

{

if (n[m].flag==1 && min>n[m].key )

{

min=n[m].key;

p=m;

}

}

n[p].flag=0;

return p;

}

void relax(int u, int v, struct node n[9])

{

if (n[v].key > n[u].key + a[u][v])

{

n[v].key= n[u].key + a[u][v];

n[v].p=u;

}

}

void main()

{

struct node n[9];

int i,j;

printf("Enter the adjacency matrix \n");

for (i=0; i<9; i++)

{

for (j=0; j<9; j++)

{

scanf("%d", &a[i][j]);

}

}

// initialize all sources

for (i=0;i<9;i++)

{

n[i].key=9999;

n[i].p=0;

n[i].flag=1;

}

n[0].key=0;

n[0].p=0;

n[0].flag=0;

for (i=0;i<9;i++)

{

int u;

u=min(n);djan

set[u]=1;

for (j=0; j<9;j++)

{

if (a[u][j]!=0)

{

relax(u,j,n);

}

}

}

printf("The Solution is \n");

printf("Key \t Parent \n");

int t;

for (t=0;t<9;t++)

{

printf("%d \t %c \n", n[t].key , 'a'+n[t].p);

}

}

// 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

/\*Output:-

student@student-HP-ProOne-400-G1-AiO:~/Desktop/101$ gcc djikstra.c -o d

student@student-HP-ProOne-400-G1-AiO:~/Desktop/101$ ./d

Enter the adjacency matrix

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

The Solution is

Key Parent

0 a

4 a

12 b

19 c

21 f

11 g

9 h

8 a

14 c

\*/

**EXPERIMENT 6**

**(Backtracking)**

**Aim: Write a program to implement N-queen problem using backtracking.**

**THEORY:**

Backtracking is finding the solution of a problem whereby the solution depends on the previous steps taken.The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. A queen can attack horizontally, vertically, or diagonally. The solution to this problem is also attempted in a similar way. We first place the first queen anywhere arbitrarily and then place the next queen in any of the safe places. We continue this process until the number of unplaced queens becomes zero (a solution is found) or no safe place is left. If no safe place is left, then we change the position of the previously placed queen.

**ALGORITHM:**

**isValid(board, row, col)**

**Input: The chess board, row and the column of the board.**

**Output:**True when placing a queen in row and place position is a valid or not.

Begin

   if there is a queen at the left of current col, then

      return false

   if there is a queen at the left upper diagonal, then

      return false

   if there is a queen at the left lower diagonal, then

      return false;

   return true //otherwise it is valid place

End

**solveNQueen(board, col)**

**Input:**The chess board, the col where the queen is trying to be placed.

**Output:**The position matrix where queens are placed.

Begin

   if all columns are filled, then

      return true

   for each row of the board, do

      if isValid(board, i, col), then

         set queen at place (i, col) in the board

         if solveNQueen(board, col+1) = true, then

            return true

         otherwise remove queen from place (i, col) from board.

   done

   return false

End

**TIME COMPLEXITY : exponential O(n!)**

**CODE:**

#include<stdio.h>

//Number of queens

int N;

//chessboard

int board[100][100];

//function to check if the cell is attacked or not

int is\_attack(int i,int j)

{

int k,l;

//checking if there is a queen in row or column

for(k=0;k<N;k++)

{

if((board[i][k] == 1) || (board[k][j] == 1))

return 1;

}

//checking for diagonals

for(k=0;k<N;k++)

{

for(l=0;l<N;l++)

{

if(((k+l) == (i+j)) || ((k-l) == (i-j)))

{

if(board[k][l] == 1)

return 1;

}

}

}

return 0;

}

int N\_queen(int n)

{

int i,j;

//if n is 0, solution found

if(n==0)

return 1;

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

{

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

{

if((!is\_attack(i,j)) && (board[i][j]!=1))

{

board[i][j] = 1;

//recursion

//wether we can put the next queen with this arrangment or not

if(N\_queen(n-1)==1)

{

return 1;

}

board[i][j] = 0;

}

}

}

return 0;

}

int main()

{

//taking the value of N

printf("Enter the value of N for NxN chessboard\n");

scanf("%d",&N);

int i,j;

//setting all elements to 0

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

{

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

{

board[i][j]=0;

}

}

//calling the function

N\_queen(N);

//printing the matix

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

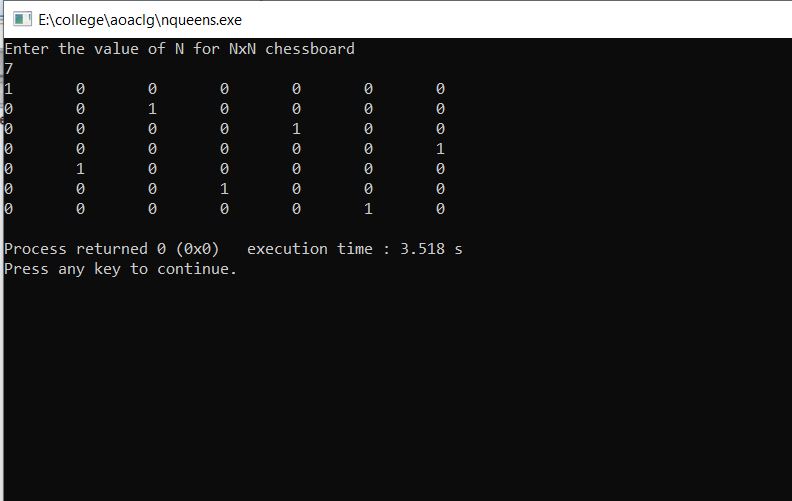
{

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

printf("%d\t",board[i][j]);

printf("\n");

}}



**Aim : To implement sum of subsets problem**

**THEORY :**

**Subset sum problem** is to find **subset** of elements that are selected from a given set whose **sum** adds up to a given number K. We are considering the set contains non-negative values. It is assumed that the input set is unique (no duplicates are presented).

**ALGORITHM:**

subsetSum(set, subset, n, subSize, total, node, sum)

**Input:** The given set and subset, size of set and subset, a total of the subset, number of elements in the subset and the given sum.

**Output:** All possible subsets whose sum is the same as the given sum.

Begin

   if total = sum, then

      display the subset

      //go for finding next subset

      subsetSum(set, subset, , subSize-1, total-set[node], node+1, sum)

      return

   else

      for all element i in the set, do

         subset[subSize] := set[i]

         subSetSum(set, subset, n, subSize+1, total+set[i], i+1, sum)

      done

End

**TIME COMPLEXITY : O(2n)**

**CODE:**

#include<stdio.h>

#include<stdlib.h>

void sumOfSub(int,int,int);

static int m=0;

int\*w;

int\*x;

void main()

{ int i=0,sum=0,n=0;

printf("Enter size of array: ");

scanf("%d",&n);

w=(int\*)malloc(sizeof(int)\*n+1);

x=(int\*)malloc(sizeof(int)\*n+1);

printf("Enter %d elements: ",n);

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

{

scanf("%d",&w[i]);

sum+=w[i];

x[i]=0;

}

printf("Enter the sum to be obtained: ");

scanf("%d",&m);

if(sum < m)

{

printf("Not possible to obtain any subset !!! ");

exit(1);

}

printf("Possible Subsets are( 0 indicates exclusion and 1 indicates inclusion) : ");

sumOfSub(0,1,sum);

}

void sumOfSub(int s,int k,int r)

{ int i=0;

x[k]=1;

if(s+w[k]==m)

{ printf("\n");

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

printf("\t%d",x[i]);

}

else if((s+w[k]+w[k+1]) <= m)

{

sumOfSub(s+w[k],k+1,r-w[k]);

}

if((s+r-w[k]) >= m && (s+w[k+1]) <= m)

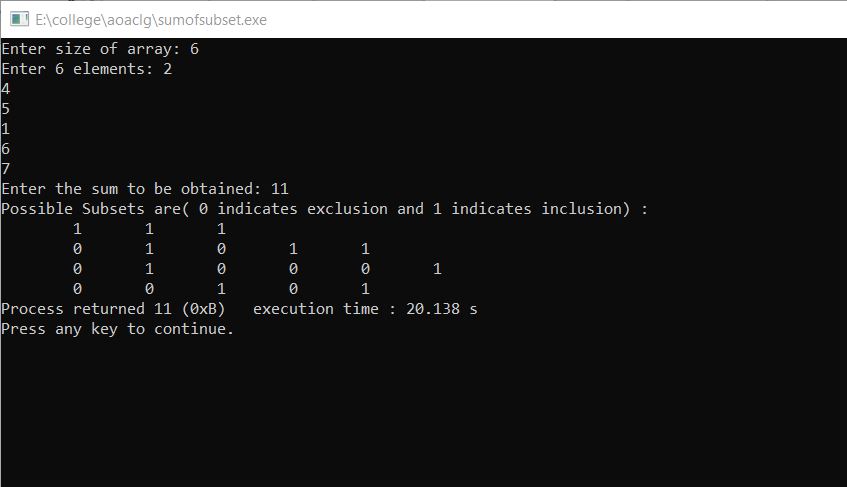
{

x[k]=0;

sumOfSub(s,k+1,r-w[k]);

}

}



**Aim: To implement the graph coloring problem**

**THEORY:**

Given an undirected graph and a number m, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with the same color. Here coloring of a graph means the assignment of colors to all vertices.

Input:  
1) A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.  
2) An integer m which is the maximum number of colors that can be used.

Output:  
An array color[V] that should have numbers from 1 to m. color[i] should represent the color assigned to the ith vertex. The code should also return false if the graph cannot be colored with m colors.

**ALGORITHM:**

**isValid(vertex, colorList, col)**

**Input:**Vertex, colorList to check, and color, which is trying to assign.

**Output:** True if the color assigning is valid, otherwise false.

Begin

   for all vertices v of the graph, do

      if there is an edge between v and i, and col = colorList[i], then

         return false

   done

   return true

End

**graphColoring(colors, colorList, vertex)**

**Input:**Most possible colors, the list for which vertices are colored with which color, and the starting vertex.

**Output:** True, when colors are assigned, otherwise false.

Begin

   if all vertices are checked, then

      return true

   for all colors col from available colors, do

      if isValid(vertex, color, col), then

         add col to the colorList for vertex

         if graphColoring(colors, colorList, vertex+1) = true, then

            return true

         remove color for vertex

   done

   return false

End

**COMPLEXITY:** The number of anode increases exponentially at every level in state space tree. With M colors and n vertices , total number of nodes in state space tree would be

1 + M + M2 + M3………Mn

T(n) = O(Mn)

**CODE:**

#include<stdio.h>

int G[50][50],x[50]; //G:adjacency matrix,x:colors

void next\_color(int k){

int i,j;

x[k]=1; //coloring vertex with color1

for(i=0;i<k;i++){ //checking all k-1 vertices-backtracking

if(G[i][k]!=0 && x[k]==x[i]) //if connected and has same color

x[k]=x[i]+1; //assign higher color than x[i]

}

}

int main(){

int n,e,i,j,k,l;

printf("Enter no. of vertices : ");

scanf("%d",&n); //total vertices

printf("Enter no. of edges : ");

scanf("%d",&e); //total edges

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

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

G[i][j]=0; //assign 0 to all index of adjacency matrix

printf("Enter indexes where value is 1-->\n");

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

scanf("%d %d",&k,&l);

G[k][l]=1;

G[l][k]=1;

}

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

next\_color(i); //coloring each vertex

printf("Colors of vertices -->\n");

for(i=0;i<n;i++) //displaying color of each vertex

printf("Vertex[%d] : %d\n",i+1,x[i]);

return 0;

}



**Aim : To implement the 15 puzzle problem**

**THEORY:**

Given a 4×4 board with 15 tiles (every tile has one number from 1 to 15) and one empty space. The objective is to place the numbers on tiles in order using the empty space. We can slide four adjacent (left, right, above and below) tiles into the empty space.

**CODE:**

#include<stdio.h>

#include<conio.h>

int m=0,n=4;

int cal(int temp[10][10],int t[10][10])

{

int i,j,m=0;

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

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

{

if(temp[i][j]!=t[i][j])

m++;

}

return m;

}

int check(int a[10][10],int t[10][10])

{

int i,j,f=1;

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

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

if(a[i][j]!=t[i][j])

f=0;

return f;

}

void main()

{

int p,i,j,n=4,a[10][10],t[10][10],temp[10][10],r[10][10];

int m=0,x=0,y=0,d=1000,dmin=0,l=0;

printf("\nEnter the matrix to be solved,space with zero :\n");

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

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

scanf("%d",&a[i][j]);

printf("\nEnter the target matrix,space with zero :\n");

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

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

scanf("%d",&t[i][j]);

printf("\nEntered Matrix is :\n");

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

{

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

printf("%d\t",a[i][j]);

printf("\n");

}

printf("\nTarget Matrix is :\n");

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

{

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

printf("%d\t",t[i][j]);

printf("\n");

}

while(!(check(a,t)))

{

l++;

d=1000;

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

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

{

if(a[i][j]==0)

{

x=i;

y=j;

}

}

//To move upwards

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

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

temp[i][j]=a[i][j];

if(x!=0)

{

p=temp[x][y];

temp[x][y]=temp[x-1][y];

temp[x-1][y]=p;

}

m=cal(temp,t);

dmin=l+m;

if(dmin < d)

{

d=dmin;

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

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

r[i][j]=temp[i][j];

}

//To move downwards

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

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

temp[i][j]=a[i][j];

if(x!=n-1)

{

p=temp[x][y];

temp[x][y]=temp[x+1][y];

temp[x+1][y]=p;

}

m=cal(temp,t);

dmin=l+m;

if(dmin < d)

{

d=dmin;

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

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

r[i][j]=temp[i][j];

}

//To move right side

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

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

temp[i][j]=a[i][j];

if(y!=n-1)

{

p=temp[x][y];

temp[x][y]=temp[x][y+1];

temp[x][y+1]=p;

}

m=cal(temp,t);

dmin=l+m;

if(dmin < d)

{

d=dmin;

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

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

r[i][j]=temp[i][j];

}

//To move left

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

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

temp[i][j]=a[i][j];

if(y!=0)

{

p=temp[x][y];

temp[x][y]=temp[x][y-1];

temp[x][y-1]=p;

}

m=cal(temp,t);

dmin=l+m;

if(dmin < d)

{

d=dmin;

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

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

r[i][j]=temp[i][j];

}

printf("\nCalculated Intermediate Matrix Value :\n");

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

{

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

printf("%d\t",r[i][j]);

printf("\n");

}

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

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

{

a[i][j]=r[i][j];

temp[i][j]=0;

}

printf("Minimum cost : %d\n",d);

}

getch();

}

/\*

output:

Enter the matrix to be solved,space with zero:

1

2

3

4

5

6

0

8

9

10

7

11

13

14

15

12

Enter the target matrix,space with zero :

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

0

Entered Matrix is :

1 2 3 4

5 6 0 8

9 10 7 11

13 14 15 12

Target Matrix is :

1 2 3 4

5 6 7 8

9 10 11 12

13 14 15 0

Calculated Intermediate Matrix Value :

1 2 3 4

5 6 7 8

9 10 0 11

13 14 15 12

Minimum cost : 4

Calculated Intermediate Matrix Value :

1 2 3 4

5 6 7 8

9 10 11 0

13 14 15 12

Minimum cost : 4

Calculated Intermediate Matrix Value :

1 2 3 4

5 6 7 8

9 10 11 12

13 14 15 0

Minimum cost : 3

\*/

**EXPERIMENT 7**

**(BRANCH AND BOUND)**

**Aim : To implement Travelling Salesman Problem**

**THEORY:**

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns back to the starting point.  
Note the difference between Hamiltonian cycle and TSP. The Hamiltoninan cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.

**ALGORITHM:**

**Traveling-Salesman-Problem**

C ({1}, 1) = 0

for s = 2 to n do

for all subsets S Є {1, 2, 3, … , n} of size s and containing 1

C (S, 1) = ∞

for all j Є S and j ≠ 1

C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}

Return minj C ({1, 2, 3, …, n}, j) + d(j, i)

**COMPLEXITY: O(**n2∗2n)

**CODE:**

int ary[10][10],completed[10],n,cost=0;

void takeInput()

{

int i,j;

printf("Enter the number of villages: ");

scanf("%d",&n);

printf("\nEnter the Cost Matrix\n");

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

{

printf("\nEnter Elements of Row: %d\n",i+1);

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

scanf("%d",&ary[i][j]);

completed[i]=0;

}

printf("\n\nThe cost list is:");

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

{

printf("\n");

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

printf("\t%d",ary[i][j]);

}

}

void mincost(int city)

{

int i,ncity;

completed[city]=1;

printf("%d--->",city+1);

ncity=least(city);

if(ncity==999)

{

ncity=0;

printf("%d",ncity+1);

cost+=ary[city][ncity];

return;

}

mincost(ncity);

}

int least(int c)

{

int i,nc=999;

int min=999,kmin;

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

{

if((ary[c][i]!=0)&&(completed[i]==0))

if(ary[c][i]+ary[i][c] < min)

{

min=ary[i][0]+ary[c][i];

kmin=ary[c][i];

nc=i;

}

}

if(min!=999)

cost+=kmin;

return nc;

}

int main()

{

takeInput();

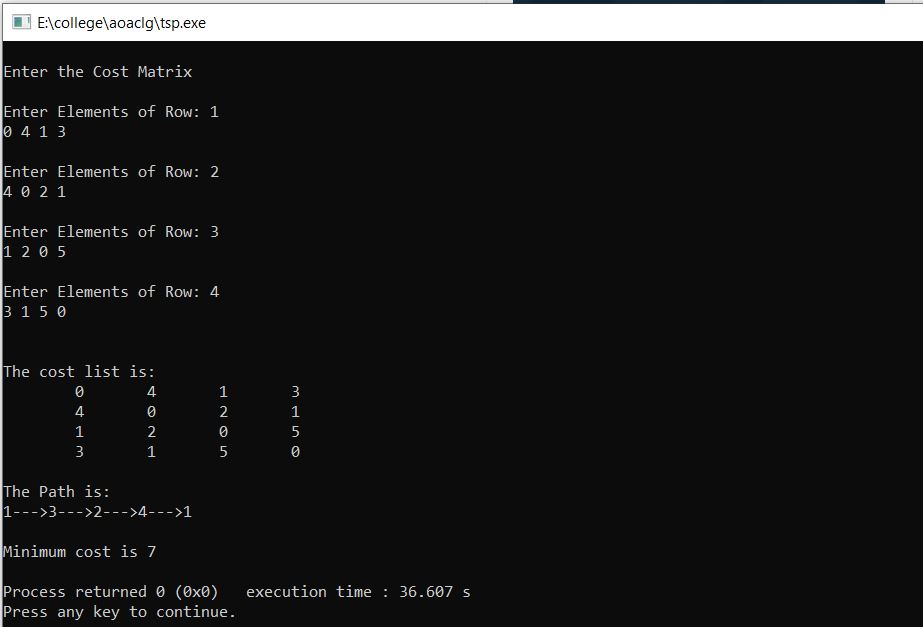
printf("\n\nThe Path is:\n");

mincost(0); //passing 0 because starting vertex

printf("\n\nMinimum cost is %d\n ",cost);

return 0;

}



**EXPERIMENT 8**

**Aim: To implement Naïve String Matching Algorithm**

**THEORY:**

The naïve approach tests all the possible placement of Pattern P [1.......m] relative to text T [1......n]. We try shift s = 0, 1.......n-m, successively and for each shift s. Compare T [s+1.......s+m] to P [1......m].

The naïve algorithm finds all valid shifts using a loop that checks the condition P [1.......m] = T [s+1.......s+m] for each of the n - m +1 possible value of s.

**ALGORITHM:**

**NAIVE-STRING-MATCHER (T, P)**

1. n ← length [T]

2. m ← length [P]

3. for s ← 0 to n -m

4. do if P [1.....m] = T [s + 1....s + m]

5. then print "Pattern occurs with shift" s

**COMPLEXITY:** The number of comparisons in best case is O(n).

The number of comparisons in the worst case is O(m\*(n-m+1)).

**CODE:**

#include <stdio.h>

#include <string.h>

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

{

int M = strlen(pat);

int N = strlen(txt);

/\* A loop to slide pat[] one by one \*/

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

int j;

/\* For current index i, check for pattern match \*/

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

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

break;

if (j == M) // if pat[0...M-1] = txt[i, i+1, ...i+M-1]

printf("Pattern found at index %d \n", i);

}

}

/\* Driver program to test above function \*/

int main()

{

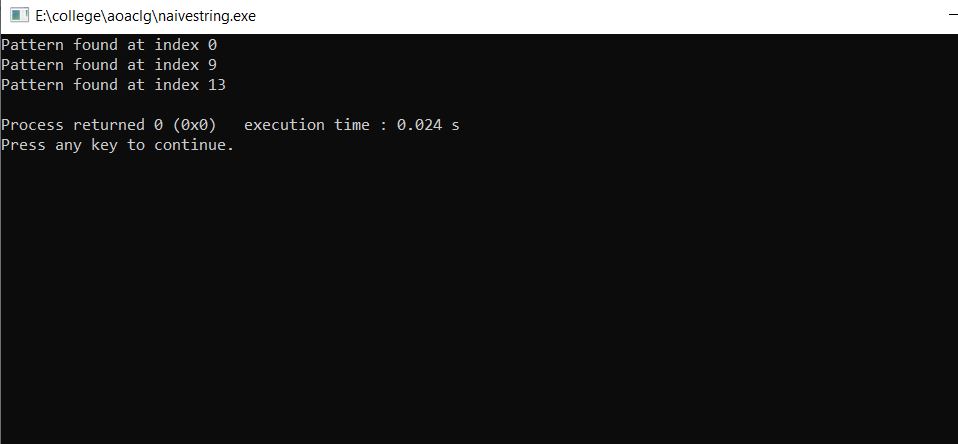
char txt[] = "AABAACAADAABAAABAA";

char pat[] = "AABA";

search(pat, txt);

return 0;

}



**Aim: To implement knuth-Morris-Pratt Algorithm**

**THEORY:**

Knuth-Morris and Pratt introduce a linear time algorithm for the string matching problem. A matching time of O (n) is achieved by avoiding comparison with an element of 'S' that have previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs

## **Components of KMP Algorithm:**

**1. The Prefix Function (Π):** The Prefix Function, Π for a pattern encapsulates knowledge about how the pattern matches against the shift of itself. This information can be used to avoid a useless shift of the pattern 'p.' In other words, this enables avoiding backtracking of the string 'S.'

**2. The KMP Matcher:** With string 'S,' pattern 'p' and prefix function 'Π' as inputs, find the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrences are found.

**ALGORITHM:**

**KMP-MATCHER (T, P)**

1. n ← length [T]

2. m ← length [P]

3. Π← COMPUTE-PREFIX-FUNCTION (P)

4. q ← 0 // numbers of characters matched

5. for i ← 1 to n // scan S from left to right

6. do while q > 0 and P [q + 1] ≠ T [i]

7. do q ← Π [q] // next character does not match

8. If P [q + 1] = T [i]

9. then q ← q + 1 // next character matches

10. If q = m // is all of p matched?

11. then print "Pattern occurs with shift" i - m

12. q ← Π [q] // look for the next match

**COMPUTE- PREFIX- FUNCTION (P)**

1. m ←length [P] //'p' pattern to be matched

2. Π [1] ← 0

3. k ← 0

4. for q ← 2 to m

5. do while k > 0 and P [k + 1] ≠ P [q]

6. do k ← Π [k]

7. If P [k + 1] = P [q]

8. then k← k + 1

9. Π [q] ← k

10. Return Π

**COMPLEXITY: The time complexity of KMP algorithm is O(n) in the worst case.**

**CODE:**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

/\*

\* finds the position of the pattern in the given target string

\* target - str, patter - word

\*/

int kmpSearch(char \*str, char \*word, int \*ptr) {

int i = 0, j = 0;

while ((i + j) < strlen(str)) {

/\* match found on the target and pattern string char \*/

if (word[j] == str[i + j]) {

if (j == (strlen(word) - 1)) {

printf("%s located at the index %d\n",

word, i + 1);

return;

}

j = j + 1;

} else {

i = i + j - ptr[j];

if (ptr[j] > -1) {

j = ptr[j];

} else {

j = 0;

}

}

}

}

void findOverlap(char \*word, int \*ptr) {

int i = 2, j = 0, len = strlen(word);

ptr[0] = -1;

ptr[1] = 0;

while (i < len) {

if (word[i - 1] == word[j]) {

j = j + 1;

ptr[i] = j;

i = i + 1;

} else if (j > 0) {

j = ptr[j];

} else {

ptr[i] = 0;

i = i + 1;

}

}

return;

}

int main() {

char word[256], str[1024];;

int \*ptr, i;

/\* get the target string from the user \*/

printf("Enter your target string:");

fgets(str, 1024, stdin);

str[strlen(str) - 1] = '\0';

printf("Enter your pattern string:");

fgets(word, 256, stdin);

word[strlen(word) - 1] = '\0';

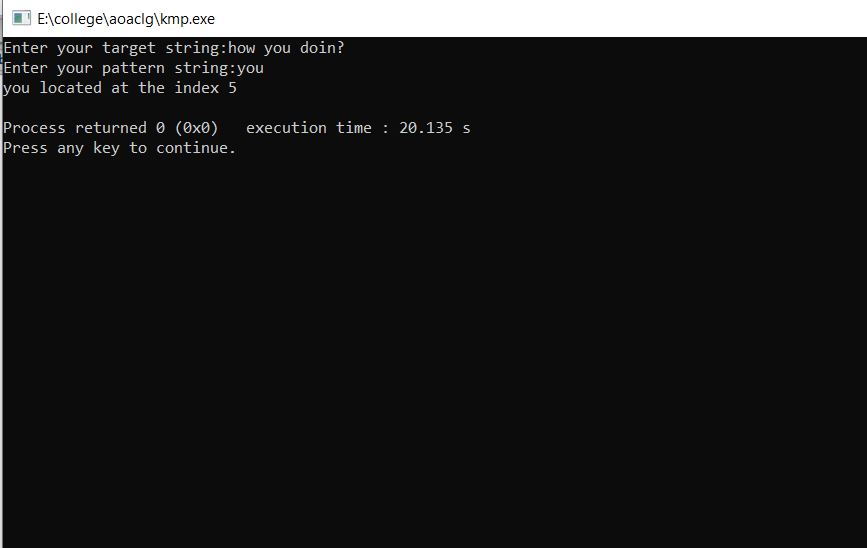
ptr = (int \*)calloc(1, sizeof(int) \* (strlen(word)));

findOverlap(word, ptr);

kmpSearch(str, word, ptr);

return 0;

}



**Aim: To implement Rabin-Karp Algorithm**

**THEORY:**

The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character subsequences of text to be compared. If the hash values are unequal, the algorithm will determine the hash value for next M-character sequence. If the hash values are equal, the algorithm will analyze the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and character matching is only required when the hash values match.

**ALGORITHM:**

**RABIN-KARP-MATCHER (T, P, d, q)**

1. n ← length [T]

2. m ← length [P]

3. h ← dm-1 mod q

4. p ← 0

5. t0 ← 0

6. for i ← 1 to m

7. do p ← (dp + P[i]) mod q

8. t0 ← (dt0+T [i]) mod q

9. for s ← 0 to n-m

10. do if p = ts

11. then if P [1.....m] = T [s+1.....s + m]

12. then "Pattern occurs with shift" s

13. If s < n-m

14. then ts+1 ← (d (ts-T [s+1]h)+T [s+m+1])mod q

**COMPLEXITY: The running time of RABIN-KARP-MATCHER in the worst case scenario O ((n-m+1) m but it has a good average case running time. If the expected number of strong shifts is small O (1) and prime q is chosen to be quite large, then the Rabin-Karp algorithm can be expected to run in time O (n+m) plus the time to require to process spurious hits.**

**CODE:**

#include<stdio.h>

#include<string.h>

int main (){

char txt[80], pat[80];

int q;

printf ("Enter the container string \n");

scanf ("%s", &txt);

printf ("Enter the pattern to be searched \n");

scanf ("%s", &pat);

int d = 256;

printf ("Enter a prime number \n");

scanf ("%d", &q);

int M = strlen (pat);

int N = strlen (txt);

int i, j;

int p = 0;

int t = 0;

int h = 1;

for (i = 0; i < M - 1; i++)

h = (h \* d) % q;

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

p = (d \* p + pat[i]) % q;

t = (d \* t + txt[i]) % q;

}

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

if (p == t){

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

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

break;

}

if (j == M)

printf ("Pattern found at index %d \n", i);

}

if (i < N - M){

t = (d \* (t - txt[i] \* h) + txt[i + M]) % q;

if (t < 0)

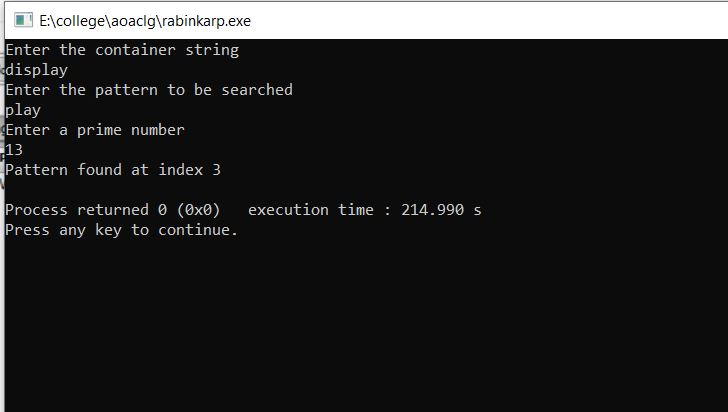
t = (t + q);

}

}

return 0;

}



**EXPERIMENT 9**

**Aim : To implement string matching with finite automata**

**THEORY:**

The string-matching automaton is a very useful tool which is used in string matching algorithm. It examines every character in the text exactly once and reports all the valid shifts in O (n) time. The goal of string matching is to find the location of specific text pattern within the larger body of text (a sentence, a paragraph, a book, etc.)

**ALGORITHM:**

**FINITE- AUTOMATON-MATCHER (T,δ, m),**

1. n ← length [T]

2. q ← 0

3. for i ← 1 to n

4. do q ← δ (q, T[i])

5. If q =m

6. then s←i-m

7. print "Pattern occurs with shift s" s

**COMPUTE-TRANSITION-FUNCTION (P, ∑)**

1. m ← length [P]

2. for q ← 0 to m

3. do for each character a ∈ ∑\*

4. do k ← min (m+1, q+2)

5. repeat k←k-1

6. Until

7. δ(q,a)←k

8. Return δ

**CODE:**

#include <stdio.h>

#include <string.h>

#define NO\_OF\_CHARS 256

void computeTransFun(char\* pat, int M, int TF[][NO\_OF\_CHARS])

{

int i, lps = 0, x;

for (x = 0; x < NO\_OF\_CHARS; x++)

TF[0][x] = 0;

TF[0][pat[0]] = 1;

for (i = 1; i <= M; i++) {

for (x = 0; x < NO\_OF\_CHARS; x++)

TF[i][x] = TF[lps][x];

// Update the entry corresponding to this character

TF[i][pat[i]] = i + 1;

// Update lps for next row to be filled

if (i < M)

lps = TF[lps][pat[i]];

}}

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

{

int M = strlen(pat);

int N = strlen(txt);

int TF[M + 1][NO\_OF\_CHARS];

computeTransFun(pat, M, TF);

int i, j = 0;

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

j = TF[j][txt[i]];

if (j == M) {

printf("\n pattern found at index %d", i - M + 1);

}}}

int main()

{

char\* txt = "GEEKS FOR GEEKS";

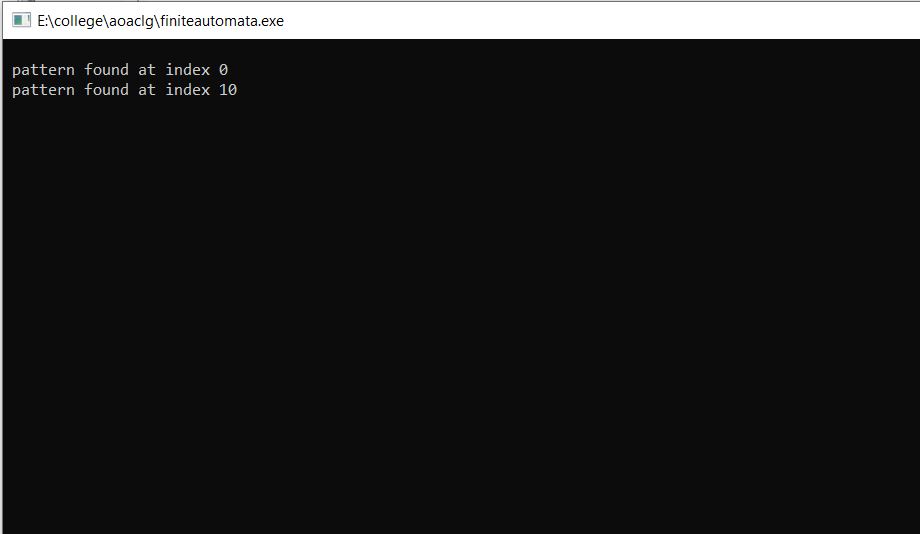
char\* pat = "GEEKS";

search(pat, txt);

getchar();

return 0;

}



**EXPERIMENT 10**

**Aim** - Case study on analysis

A) Consider or find one scenario where analysis of algorithm helped to improve to existing systems and improved time and space complexity of the previous system to get an improved one.

B) Understand and write algorithm for the same technique.

Question – To find the number of symmetric pairs among ‘n’ given pairs.  
Eg – (1, 2) (2, 3) (7, 9) (10, 9) (9, 10) (3, 2)

The number of symmetric pairs for the above example is – **2** i.e (2, 3) & (10,9)

If we use the naïve approach, then the algorithm is as follows:-

1. START
2. Input ‘n’ symmetric pairs
3. for each pair check if the its symmetric pair exists in the remaining pairs
4. EXIT

It is obvious that the above algorithm would require nesting of ‘for’ loops in which the first one would run for ‘n’ cases while the nested ‘for’ would run for ‘n-1’ cases thereby leading to time complexity of **O(n­­­2)**.

A much more efficient method would be using hashing or a hash table. The first part of the pair would be the key while the second part would be the element. Hence while traversing through the hash table, if the key is empty, then we fill it with the element and then further, while traversing, check for its symmetric pair whether the same key and element are corresposding. The algorithm would be as follows:-

1. START
2. Input ‘n’ symmetric pairs
3. for i is 0 to n
4. if (hash [array [t][1] ] != array[t][0])
   1. hash [array [t][1]] = array[t][0];
   2. hash [array[t][0]] = array[t][1];
5. else
   1. print ("Symmetric Pair - array[t][0],array[t][1])
6. EXIT

As we can see from the above algorithm, we need to traverse through the symmetric pairs only once, hence the time complexity of the alogorithm is

reduced to **O(n)** only. However the space complexity might increase due to the use of the hash table.