1.Implement Euclid's, Consecutive integer checking and Modified Euclid's algorithms to find GCD of two nonnegative integers and perform comparative analysis by generating best case and worst case data.

TESTER

```
#include<stdio.h>
#include<stdlib.h>
#define x 10
#define y 100

int test=0;

float euclid(int m,int n)
{
int r;
float count=0;
while(n)
{
count++;
r=m%n;
m=n;
n=r;
}
printf("THE GCD IS %d\n",m);
return count;
}
float consec(int m, int n)
{
int min;
```

```
float count=0;
min=m;
if(n<min)
min=n;
while(1)
{
count++;
if(m%min==0)
{
count++;
if(n%min==0)
break;
min-=1;
}
else
min-=1;
}

printf("THE GCD IS %d\n",min);
return count;
}
float modified(int m,int n)
{
int temp;
float count=0;
while(n>0)
{
```

```
if(n>m)
{
temp=m;m=n;n=temp;
}
m=m-n;
count +=1 ;
}

printf("THE GCD IS %d\n",m);
return count; // m is the GCD
}



void main()
{
int ch;
while(1)
{
printf("GCD\n");
printf("1.Euclid\n2.modified Euclid\n3.consecutive integer method\n0to exit\n");
scanf("%d",&ch);
if(ch==0)
break;
printf("ENTER THE VALUES M AND N\n");
int m,n;
scanf("%d",&m);
scanf("%d",&n);
```

```
switch(ch)
{
case 1:euclid(m,n);break;
case 2:modified(m,n);break;
case 3:consec(m,n);break;
default:break;
}
}
}
```

THIS IS FOR PLOTTER

```
#include<stdio.h>
#include<stdlib.h>
#define x 10
#define y 100
int test=0;
float euclid(int m,int n)
{
int r;
float count=0;
while(n)
{
count++;
r=m%n;
m=n;
n=r;
}
```

```
return count;
}
float consec(int m, int n)
{
int min;
float count=0;
min=m;
if(n<min)
min=n;
while(1)
{
count++;
if(m%min==0)
{
count++;
if(n%min==0)
break;
min-=1;
}
else
min-=1;
}
return count;
}
float modified(int m,int n)
{
int temp;
```

```
float count=0;
while(n>0)
{
if(n>m)
{
temp=m;m=n;n=temp;
}
m=m-n;
count +=1 ;
}

return count; // m is the GCD
}
void analysis(int ch)
{
int m,n,i,j,k;
float count,maxcount,mincount;
FILE *fp1,*fp2;
for(i=x;i<=y;i+=10)
{
maxcount=0; mincount=10000;
for (j=2;j<=i; j++ ) // To generate the data
{
for(k=2;k<=i; k++)
{
count=0;
m=j;
```

```
n=k;
switch(ch)
{
case 1:count=euclid(m,n);
break;
case 2:count=consec(m,n);
break;
case 3:count=modified(m,n);
break;
}
if(count>maxcount) // To find the maximum basic operations among all the combinations between 2 to n
maxcount=count;
if(count<mincount)
// To find the minimum basic operations among all the combinations between 2 to n
mincount=count;
}
}
switch(ch)
{
case 1:fp2=fopen("e_b.txt","a");
fp1=fopen("e_w.txt","a");
break;
case 2:fp2=fopen("c_b.txt","a");
fp1=fopen("c_w.txt","a");
break;
case 3:fp2=fopen("m_b.txt","a");
```

```
fp1=fopen("m_w.txt","a");
break;
}
fprintf(fp2,"%d %.2f\n",i,mincount);
fclose(fp2);
fprintf(fp1,"%d %.2f\n",i,maxcount);
fclose(fp1);
}
}
void main()
{
int ch;
while(1)
{
printf("GCD\n");
printf("1.Euclid\n3.modified Euclid\n2.consecutive integer method\n0 to exit\n");
scanf("%d",&ch);
if(ch ==0)
break;
switch(ch)
{
case 1:
case 2:
case 3: analysis(ch);
break;
default:break;
}
```

}

return ;

}

OUTPUT

TESTER

```
GCD
1.Euclid
2.modified Euclid
3.consecutive integer method
0to exit
1
ENTER THE VALUES M AND N
10 20
THE GCD IS 10
GCD
1.Euclid
2.modified Euclid
3.consecutive integer method
0to exit
2
ENTER THE VALUES M AND N
2 8
THE GCD IS 2
GCD
1.Euclid
2.modified Euclid
3.consecutive integer method
0to exit
3
```

```
ENTER THE VALUES M AND N
2 4
THE GCD IS 2
GCD
1.Euclid
2.modified Euclid
3.consecutive integer method
0to exit
0
```

PLOTTER

EUCLIDS:

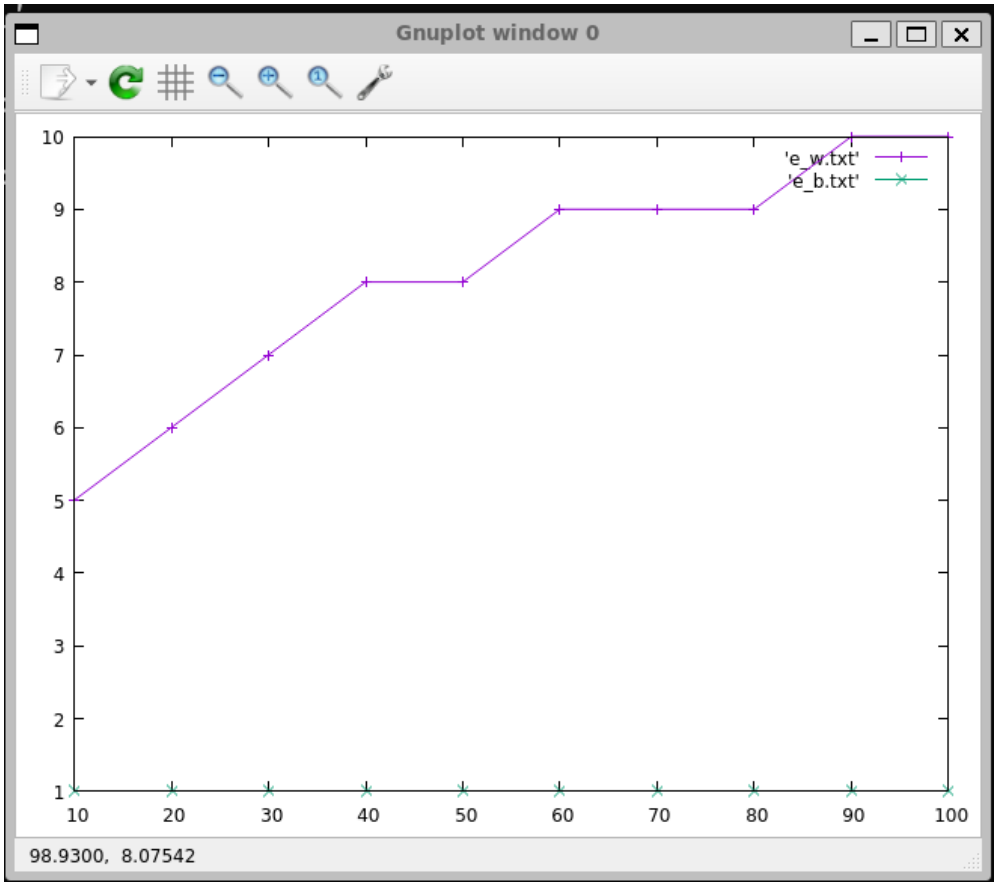
BEST CASE:

```
 1 10   1.00
 2 20   1.00
 3 30   1.00
 4 40   1.00
 5 50   1.00
 6 60   1.00
 7 70   1.00
 8 80   1.00
 9 90   1.00
10 100  1.00
```

WORST CASE:


```
 1 10 5.00
 2 20 6.00
 3 30 7.00
 4 40 8.00
 5 50 8.00
 6 60 9.00
 7 70 9.00
 8 80 9.00
 9 90 10.00
10 100 10.00
```

GRAPH:



MODIFIED EUCLIDS

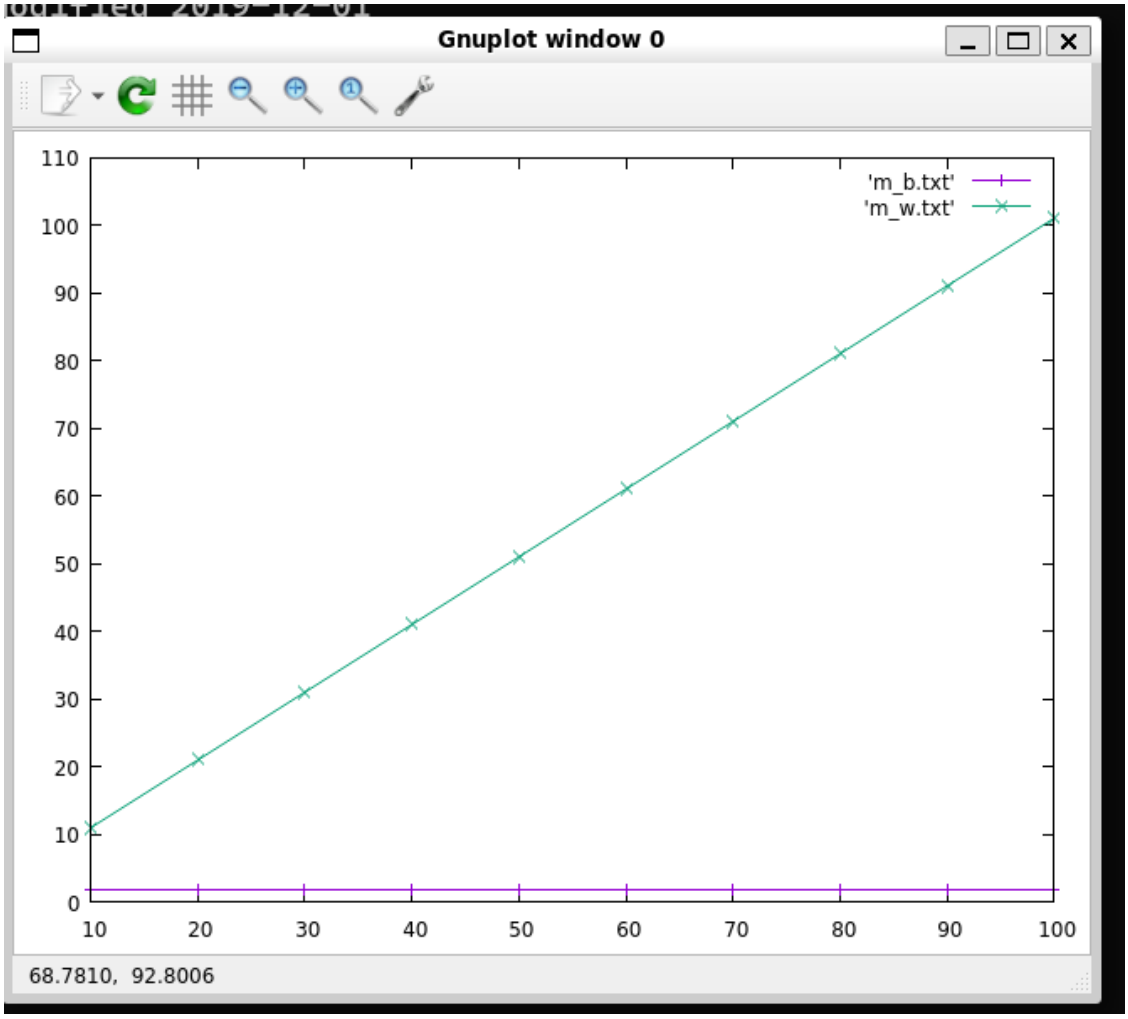
BEST CASE:

```
1 10 2.00
2 20 2.00
3 30 2.00
4 40 2.00
5 50 2.00
6 60 2.00
7 70 2.00
8 80 2.00
9 90 2.00
10 100 2.00
```

WORST CASE:

```
1 10 11.00
2 20 21.00
3 30 31.00
4 40 41.00
5 50 51.00
6 60 61.00
7 70 71.00
8 80 81.00
9 90 91.00
10 100 101.00
```

GRAPH:



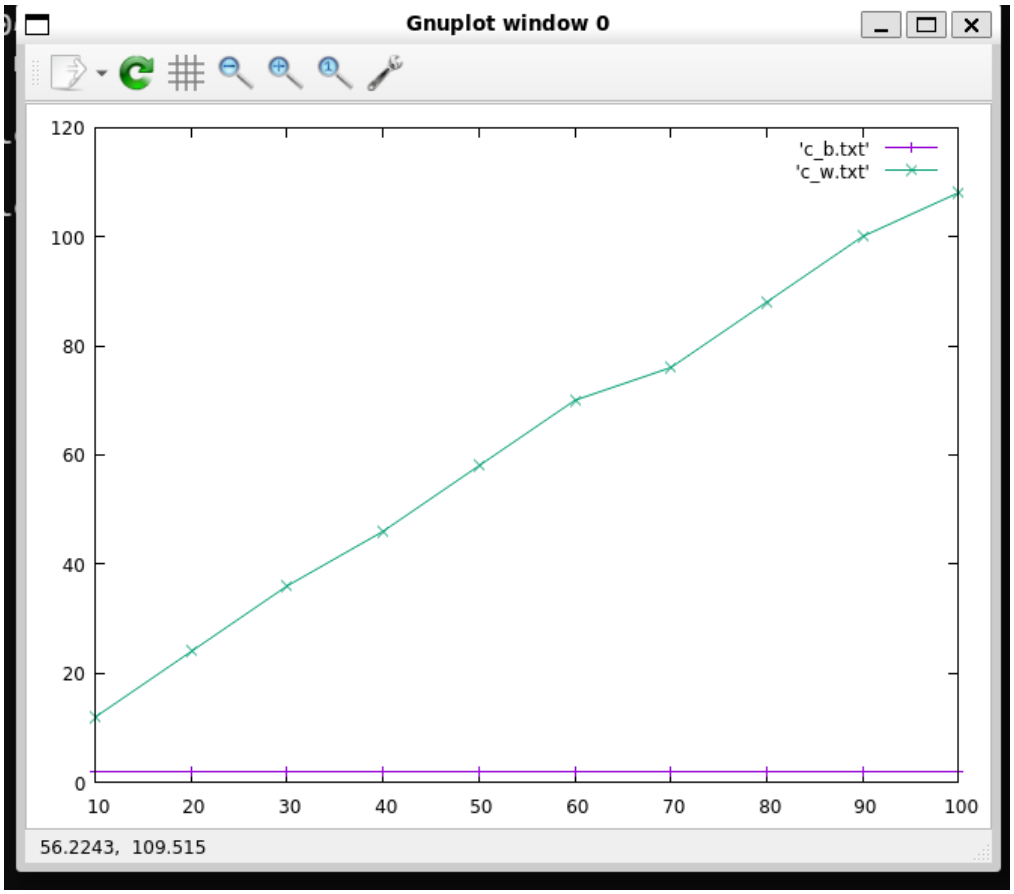
CONSECTIVE INTEGER:

BEST CASE:

```
1 10 2.00
2 20 2.00
3 30 2.00
4 40 2.00
5 50 2.00
6 60 2.00
7 70 2.00
8 80 2.00
9 90 2.00
10 100 2.00
```

WORST CASE:

```
 1 10 12.00
 2 20 24.00
 3 30 36.00
 4 40 46.00
 5 50 58.00
 6 60 70.00
 7 70 76.00
 8 80 88.00
 9 90 100.00
10 100 108.00
```


Gnuplot window 0
'c_b.txt'
'c_w.txt'
120
100
80
60
40
20
0
10
20
30
40
50
60
70
80
90
100
56.2243, 109.515

2. Implement the following searching algorithms and perform their analysis for worst case, best case and average case.

a.Sequential search b.Binary search(Recursive)

TESTER

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int linearSearch(int *a, int k, int n) {
    for (int i = 0; i < n; i++) {
        if (*(a + i) == k) {
            return i;
        }
    }
    return -1;
}

int binarySearch(int key, int *a, int high, int low) {
    if (low <= high) {
        int mid = low + (high - low) / 2; // To avoid potential overflow
        if (*(a + mid) == key)
            return mid;
        else if (*(a + mid) > key)
            return binarySearch(key, a, mid - 1, low);
        else
            return binarySearch(key, a, high, mid + 1);
    }
    return -1;
}
```

```
int main() {
   int arr[100];
   int n, key, r;


   for (;;) {
      printf("ENTER 1. TO LINEAR SEARCH\n2. TO BINARY SEARCH\n3.
TO EXIT\n");
      int ch;
      scanf("%d", &ch);


      switch (ch) {
         case 1:
            printf("ENTER THE NUMBER OF ELEMENTS\n");
            scanf("%d", &n);
            printf("ENTER THE ELEMENTS OF THE ARRAY\n");
            for (int i = 0; i < n; i++) {
               scanf("%d", &arr[i]);
            }
            printf("ENTER THE KEY ELEMENT\n");
             scanf("%d", &key);
            r = linearSearch(arr, key, n);
            if (r != -1) {
               printf("The element is present at the index %d\n", r);
            } else {
               printf("Element not found\n");
            }
            break;
```

```
        case 2:
            printf("ENTER THE NUMBER OF ELEMENTS\n");
            scanf("%d", &n);
            printf("ENTER THE ELEMENTS OF THE ARRAY\n");
            for (int i = 0; i < n; i++) {
                scanf("%d", &arr[i]);
            }
            printf("ENTER THE KEY ELEMENT\n");
            scanf("%d", &key);
            r = binarySearch(key, arr, n - 1, 0);
            if (r != -1) {
                printf("The element is present at the index %d\n", r);
            } else {
                printf("Element not found\n");
            }
            break;
        default:
            exit(0);
      }
   }
   return 0;
}
```

PLOTTER

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
int count;
int linearSearch(int *a, int k, int n)
{
   int i;
    count = 0;
   for (i = 0; i < n; i++)
   {
      count++;
      if (*(a + i) == k)
      {
         return count;
      }
   }
   return count;
}
int binarySearch(int key, int *a, int high, int low)
{
   int mid;
   count++;
   mid = (high + low) / 2;
   if (low > high)
      return count-1;
   if (*(a + mid) == key)
      return count;
   else if (*(a + mid) > key)
      return binarySearch(key,a,mid - 1,low);
```

```
    else
 return binarySearch(key, a, high, mid + 1);
}
void plotter1()
{
  srand(time(NULL));
    int *arr;
    int n,key,r;
    FILE *f1,*f2,*f3;
    f1=fopen("linearbest.txt","a");
    f2=fopen("linearavg.txt","a");
    f3=fopen("linearworst.txt","a");
         n=2;
    while(n<=1024)
    {
         arr=(int *)malloc(n*sizeof(int));
        for(int i=0;i<n;i++)
        *(arr+i)=1;
        r=linearSearch(arr,1,n);
       fprintf(f1,"%d\t%d\n",n,r);
       for (int i = 0; i < n; i++)
       *(arr+i)=rand()%n;
       key=rand()%n;
        r=linearSearch(arr,key,n);
       fprintf(f2,"%d\t%d\n",n,r);
        for(int i=0;i<n;i++)
        *(arr+i)=0;
```

```
        r=linearSearch(arr,1,n);
        fprintf(f3,"%d\t%d\n",n,r);
        n=n*2;
        free(arr);
    }
    fclose(f1);
    fclose(f2);
    fclose(f3);
}
void plotter2()
{
     srand(time(NULL));
    int *arr;
    int n,key,r;
    FILE *f1,*f2,*f3;
    f1=fopen("binarybest.txt","a");
    f2=fopen("binaryavg.txt","a");
    f3=fopen("binaryworst.txt","a");
         n=2;
    while(n<=1024)
    {
         arr=(int *)malloc(n*sizeof(int));
        for(int i=0;i<n;i++)
        *(arr+i)=1;
        int mid=(n-1)/2;
        *(arr+mid)=0;
        count=0;
```

```
        r=binarySearch(0,arr,n-1,0);
        fprintf(f1,"%d\t%d\n",n,r);
        printf("%d\t%d\n",n,count);
        for (int i = 0; i < n; i++)
        *(arr+i)=rand()%n;
        key=rand()%n+1;
        count=0;
        r=binarySearch(-1,arr,n-1,0);
        fprintf(f2,"%d\t%d\n",n,r);

        printf("%d\t%d\n",n,count);
         for(int i=0;i<n;i++)
         *(arr+i)=0;
         count=0;
         r=binarySearch(1,arr,n-1,0);
        fprintf(f3,"%d\t%d\n",n,r);

        printf("%d\t%d\n",n,count);
        n=n*2;
        free(arr);
    }
    fclose(f1);
    fclose(f2);
    fclose(f3);
}
void main()
{
```

```
    plotter1();
 plotter2();
}
```

OUTPUT

TESTER

```
ENTER 1. TO LINEAR SEARCH
2. TO BINARY SEARCH
3. TO EXIT
1
ENTER THE NUMBER OF ELEMENTS
4
ENTER THE ELEMENTS OF THE ARRAY
4 6 0 8
ENTER THE KEY ELEMENT
9
Element not found
ENTER 1. TO LINEAR SEARCH
2. TO BINARY SEARCH
3. TO EXIT
2
ENTER THE NUMBER OF ELEMENTS
5
ENTER THE ELEMENTS OF THE ARRAY
9 0 -1 2 4
ENTER THE KEY ELEMENT
-1
The element is present at the index 2
ENTER 1. TO LINEAR SEARCH
2. TO BINARY SEARCH
3. TO EXIT
3
```

PLOTTER

Linear search best case

Open

linearbest.txt
~/ADA

```
1 2      1
2 4      1
3 8      1
4 16     1
5 32     1
6 64     1
7 128    1
8 256    1
9 512    1
10 1024  1
```

Linear worst case

Open

linearworst.txt
~/ADA

```
2       2
4       4
8       8
16      16
32      32
64      64
128     128
256     256
512     512
1024    1024
```

Linear average case

Open

linearavg.txt
~/ADA

```
2       1
4       3
8       8
16      16
32      32
64      39
128     82
256     256
512     391
1024    52
```

Binary best case

Open

binarybest.txt
~/ADA

```
2       1
4       1
8       1
16      1
32      1
64      1
128     1
256     1
512     1
1024    1
```

Binary worst case

Open

binaryworst.txt
~/ADA

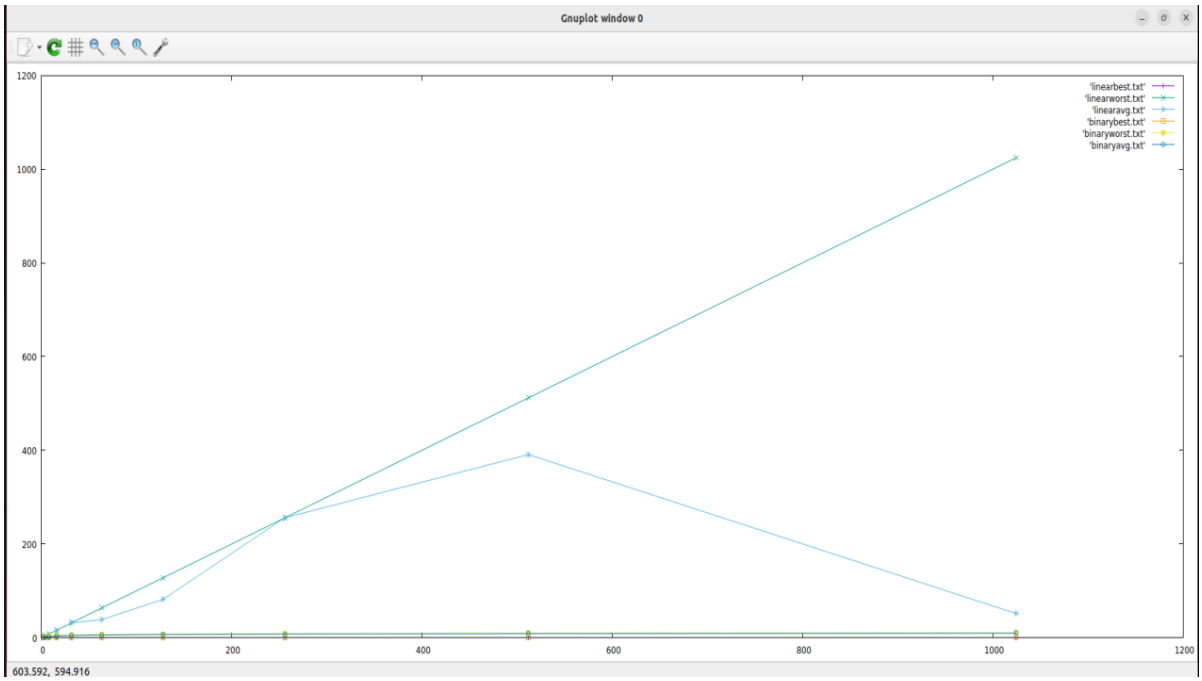
```
2       2
4       3
8       4
16      5
32      6
64      7
128     8
256     9
512     10
1024    11
```

Binary average case

Open ▾ binaryavg.txt ~/ADA

```
2       1
4       2
8       3
16      4
32      5
64      6
128     7
256     8
512     9
1024    10
```

Gnuplot



3. Implement the following elementary sorting algorithms and perform their analysis for worst case, best case and average inputs

a. Bubble sort b. Insertion sort c. Selection sort

a. PLOTTER and TESTER

```
#include<stdio.h>

#include<stdlib.h>

int count;

int bubblesort(int *a,int n)

{
```

```
    count = 0;
     int i,j,t,flag=0;
   for(i=0;i<n-1;i++)
   {
      flag=0;
    for(j=0;j<n-i-1;j++)
     {
        count++;
       if(a[j]>a[j+1])
      {
        t=*(a+j);
        *(a+j)=*(a+j+1);
        *(a+j+1)=t;
        flag=1;
      }
     }
      if(flag==0)
      break;
      }
 return count;
}
void plotter()
{
    int *arr,n;
   srand(time(NULL));
   FILE *f1,*f2,*f3;
   f1=fopen("BUBBLWBEST.txt","a");
```

```
f2=fopen("BUBBLEWORST.txt","a");
f3=fopen("BUBBLEAVG.txt","a");
 n=10;

 while(n<=30000)
 {
  arr=(int *)malloc(sizeof(int)*n);
  for(int i=0;i<n;i++)
  *(arr+i)=n-i;
   count=0;
    //wrost case
    bubblesort(arr,n);
    fprintf(f2,"%d\t%d\n",n,count);
    //printf("%d\t%d\n",n,count);

 //best case
  count=0;
  for(int i=0;i<n;i++)
  *(arr+i)=i+1;
  bubblesort(arr,n);
  fprintf(f1,"%d\t%d\n",n,count);
  //printf("%d\t%d\n",n,count);

 //AVG case
  for(int i=0;i<n;i++)
  *(arr+i)=rand()%n;
  count=0;
```

```
      bubblesort(arr,n);
      fprintf(f3,"%d\t%d\n",n,count);
      if(n<10000)
      n=n*10;
      else
      n=n+10000;
      free(arr);
    }
    fclose(f1);
    fclose(f2);
    fclose(f3);
}
void tester()
{
  int *arr, n;
   printf("ENTER THE NUMBER OF ELEMENTS\n");
   scanf("%d",&n);
    arr=(int *)malloc(sizeof(int)*n);
   printf("ENTER THE ELEMENTS OF THE ARRAY\n");
      for(int i=0;i<n;i++)
       scanf("%d",&arr[i]);

  printf("THE ELEMENTS OF THE ARRAY BEFORE SORTING\n");
    for(int i=0;i<n;i++)
       printf("%d ",arr[i]);
    printf("\n");
```

```
     bubblesort(arr,n);


     printf("THE ELEMENTS OF THE ARRAY BEFORE SORTING\n");
   for(int i=0;i<n;i++)
     printf("%d ",arr[i]);
     printf("\n");
     printf("\n");
}
void main()
{
   for(;;)
   {
      int key;
      printf("ENTER THE CHOICE \n1.TO TEST \n2.TO PLOT\nO  TO
EXIT\n");
      scanf("%d",&key);
      switch(key)
       {
         case 1:tester();break;
         case 2:plotter();break;
         default:exit(1);
       }
   }
}
```

OUTPUT

TESTER

```
ENTER THE CHOICE
1.TO TEST
2.TO PLOT
0  TO EXIT
1
ENTER THE NUMBER OF ELEMENTS
5
ENTER THE ELEMENTS OF THE ARRAY
3 4 0 -1 -2
THE ELEMENTS OF THE ARRAY BEFORE SORTING
3 4 0 -1 -2
THE ELEMENTS OF THE ARRAY BEFORE SORTING
-2 -1 0 3 4
```

PLOTTER

Best case

Open *BUBBLWBEST.txt ~/ADA

```
10      9
100     99
1000    999
10000   9999
20000   19999
30000   29999
```

Worst Case

Open *BUBBLEWORST.txt ~/ADA

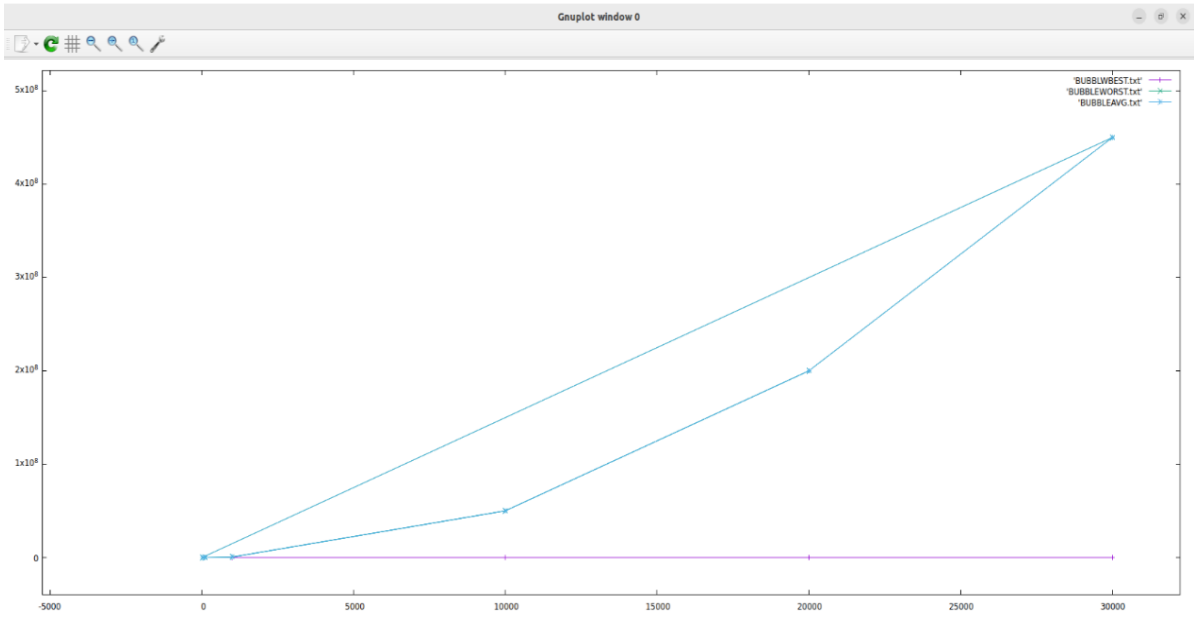
```
10      45
100     4950
1000    499500
10000   49995000
20000   199990000
30000   449985000
```

Average case

Open *BUBBLEAVG.txt ~/ADA

```
10      42
100     4859
1000    499269
10000   49994724
20000   199953685
30000   449962845
```

Gnuplot



b.

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
int count;
void insertionSort(int *arr, int n) {
   count = 0;
   for(int i=1; i<n; i++) {
      int value = arr[i];
      int j= i-1;
      while(count++ && arr[j] > value) {
         arr[j+1] = arr[j];
         j--;
         if(j<0)
            break;
      }
      arr[j+1] = value;
```

```
    }
}
void plotter()
{
    int *arr,n;
   srand(time(NULL));
   FILE *f1,*f2,*f3;

   f1=fopen("INSERTIONBEST.txt","a");
   f2=fopen("INSERTIONWORST.txt","a");
   f3=fopen("INSERTIONAVG.txt","a");
    n=10;
     while(n<=30000)
    {
        arr=(int *)malloc(sizeof(int)*n);
      for(int i=0;i<n;i++)
      *(arr+i)=n-i;
       count=0;
        //worst case
        insertionSort(arr,n);
        fprintf(f2,"%d\t%d\n",n,count);
        //printf("%d\t%d\n",n,count);

     //best case
      count=0;
      for(int i=0;i<n;i++)
      *(arr+i)=i+1;
```

```
        insertionSort(arr,n);
       fprintf(f1,"%d\t%d\n",n,count);
       //printf("%d\t%d\n",n,count);

      //AVG case
       for(int i=0;i<n;i++)
       *(arr+i)=rand()%n;
       count=0;
         insertionSort(arr,n);
       fprintf(f3,"%d\t%d\n",n,count);
       if(n<10000)
       n=n*10;
       else
       n=n+10000;
       free(arr);
     }
     fclose(f1);
     fclose(f2);
     fclose(f3);
}
void tester()
{
  int *arr, n;
    printf("ENTER THE NUMBER OF ELEMENTS\n");
    scanf("%d",&n);

    arr=(int *)malloc(sizeof(int)*n);
```

```
   printf("ENTER THE ELEMENTS OF THE ARRAY\n");
      for(int i=0;i<n;i++)
       scanf("%d",&arr[i]);

    printf("THE ELEMENTS OF THE ARRAY BEFORE SORTING\n");
     for(int i=0;i<n;i++)
       printf("%d ",arr[i]);
     printf("\n");

       insertionSort(arr,n);

      printf("THE ELEMENTS OF THE ARRAY BEFORE SORTING\n");
     for(int i=0;i<n;i++)
       printf("%d ",arr[i]);
       printf("\n");
       printf("\n");
}
void main()
{
    for(;;)
    {
        int key;
       printf("ENTER THE CHOICE \n1.TO TEST \n2.TO PLOT\nO  TO
EXIT\n");
       scanf("%d",&key);

        switch(key)
        {
```

```
        case 1:tester();break;
        case 2:plotter();break;
        default:exit(1);
      }
   }
}
```

OUTPUT

TESTER

```
ENTER THE CHOICE
1.TO TEST
2.TO PLOT
0  TO EXIT
1
ENTER THE NUMBER OF ELEMENTS
8
ENTER THE ELEMENTS OF THE ARRAY
0 4 -8 9 10 2 4 7
THE ELEMENTS OF THE ARRAY BEFORE SORTING
0 4 -8 9 10 2 4 7
THE ELEMENTS OF THE ARRAY BEFORE SORTING
-8 0 2 4 4 7 9 10
```

PLOTTER

Best case

Open

INSERTIONBEST.txt
~/ADA

```
1 10      9
2 100     99
3 1000    999
4 10000   9999
5 20000   19999
6 30000   29999
```

Worst case

Open

INSERTIONWORST.txt
~/ADA

```
1 10      45
2 100     4950
3 1000    499500
4 10000   49995000
5 20000   199990000
6 30000   449985000
```

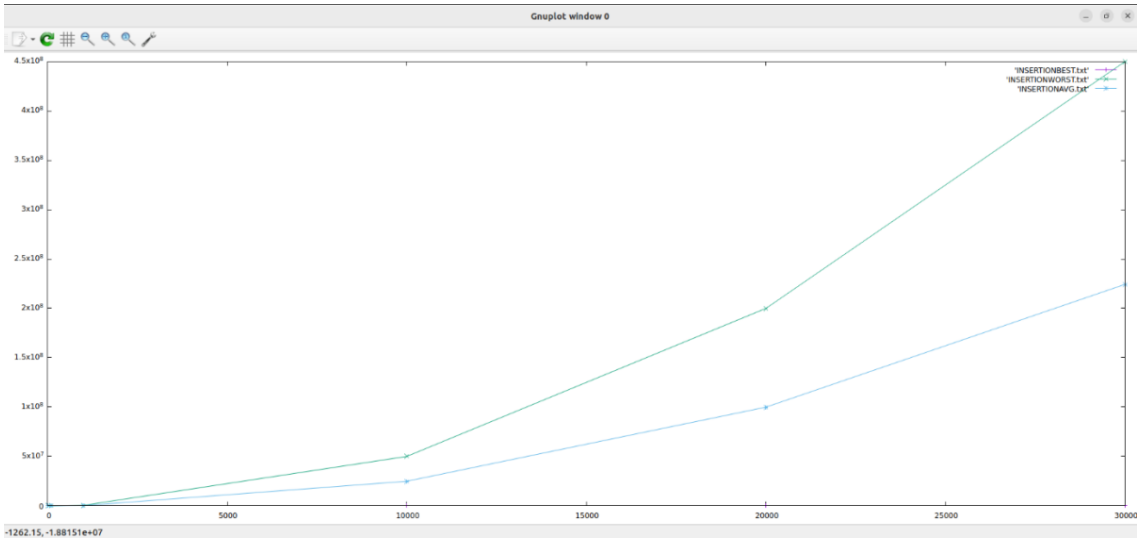
Average case

Open

INSERTIONAVG.txt
~/ADA

```
10      18
100     2385
1000    249441
10000   24851052
20000   99940578
30000   224345859
```

Gnuplot



c. #include<stdio.h>

#include<stdlib.h>

#include<time.h>

int count;

void selectionsort(int *a,int n)

{

int i,j,min,t;

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

{

min=i;

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

{

```
        if((a+j)<(a+min))
         min=j;
       count++;
     }

     if(min!=i)
     {
       t=*(a+min);
       *(a+min)=*(a+i);
       *(a+i)=t;
     }
}
}
void tester()
{
  int *arr, n;
   printf("ENTER THE NUMBER OF ELEMENTS\n");
   scanf("%d",&n);

   arr=(int *)malloc(sizeof(int)*n);
   printf("ENTER THE ELEMENTS OF THE ARRAY\n");
     for(int i=0;i<n;i++)
      scanf("%d",&arr[i]);

  printf("THE ELEMENTS OF THE ARRAY BEFORE SORTING\n");
    for(int i=0;i<n;i++)
      printf("%d ",arr[i]);
```

```
    printf("\n");

      selectionsort(arr,n);

     printf("THE ELEMENTS OF THE ARRAY BEFORE SORTING\n");
    for(int i=0;i<n;i++)
      printf("%d ",arr[i]);
      printf("\n");
      printf("\n");
}
void plotter()
{
  FILE *f;
    f=fopen("selectionsort.txt", "a");
  int j;
  int n=10;
  while (n<=30000)
  {
     int *a=(int)malloc(sizeof(int)*n);
    for(int i=0;i<n;i++)
    {
      *(a+i)=i;
    }
    count=0;
    selectionsort(a,n);
    fprintf(f,"%d\t%d\n",n,count);
    printf("%d\t%d\n",n,count);
```

```
    if(n<10000)
        n*=10;
      else
      n+=10000;

  }
}

void main()
{
   for(;;)
   {
      int key;
      printf("ENTER THE CHOICE \n1.TO TEST \n2.TO PLOT\nO  TO EXIT\n");
      scanf("%d",&key);
       switch(key)
       {
        case 1:tester();break;
        case 2:plotter();break;
        default:exit(1);
       }
   }
}
```

OUTPUT

TESTER

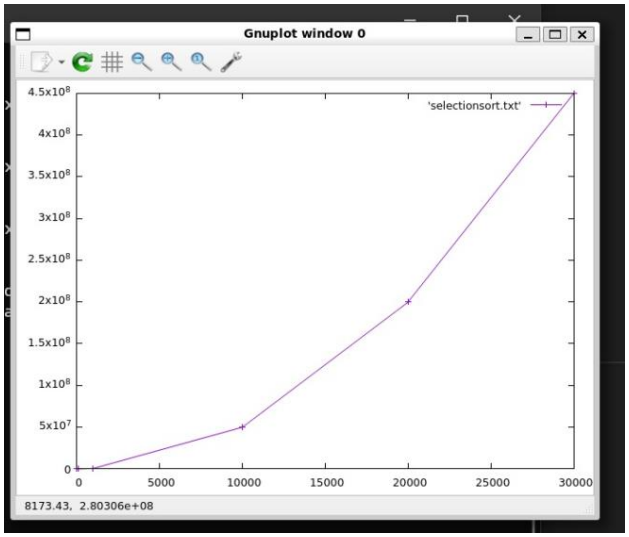
```
ENTER THE NUMBER OF ELEMENTS
5
ENTER THE ELEMENTS OF THE ARRAY
0
-5
9
1
-2
THE ELEMENTS OF THE ARRAY BEFORE SORTING
0 -5 9 1 -2
THE ELEMENTS OF THE ARRAY BEFORE SORTING
-5 -2 0 1 9
```

PLOTTER

General case

```
1 10      45
2 100     4950
3 1000    499500
4 10000   49995000
5 20000   199990000
6 30000   449985000
```

Gnuplot



4.Implement the brute force string matching algorithm to search for a pattern length 'M' in a a text of length 'N'(M<=N) and perform its analysis for worst case, best case and average inputs.

TESTER

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
int count = 0;
int stringmatching(char *text, char *pattern, int n, int m) {
    count = 0;
    for (int i = 0; i <= n - m; i++) {
       int j = 0;
       while (j < m) {
           count++;
       if (pattern[j] != text[i + j])
          break;
           j++;
  }
   if (j == m) {
         printf("THE PATTERN FOUND \n");
         return count;
     }
  }
      printf("THE PATTERN not found \n");
     return count;
}

int main() {
     int m, n;
     char text[100], pattern[100];
     printf("ENTER THE PATTERN LENGTH\n");
     scanf("%d", &m);
     printf("ENTER THE PATTERN\n");
     getchar(); // Consume the newline character left in the input buffer
```

```
    fgets(pattern, sizeof(pattern), stdin);
    pattern[strcspn(pattern, "\n")] = '\0'; // Remove the newline character from the
input
 printf("ENTER THE TEXT LENGTH\n");
 scanf("%d", &n);
 printf("ENTER THE TEXT\n");
getchar(); // Consume the newline character left in the input buffer
 fgets(text, sizeof(text), stdin);
text[strcspn(text, "\n")] = '\0'; // Remove the newline character from the input
   int comparisons = stringmatching(text, pattern, n, m);
 printf("Number of comparisons: %d\n", comparisons);
 return 0;
}
```

PLOTTER

```
//Program to perform analysis of brute force string matching
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
  int count = 0;


int stringmatching(char *text, char *pattern, int n, int m) {
     count = 0;
    for(int i=0; i<=n-m; i++)
  {
      int j=0;
      while(j<m)
     {
         count++;
```

```
            if(pattern[j] != text[i+j])
                break;  j++;
    }
        if(j==m) {
            return count;
        }
     }
       return count;
    }
    void ploter()
    {
       FILE *f1 =fopen("stringbest.txt", "a");
        FILE *f2 =fopen("stringworst.txt", "a");
       FILE *f3 =fopen("stringavg.txt", "a");
     char *text=(char *)malloc(1000*sizeof(char));
     char * pattern;
      for(int i=0;i<1000;i++)
      *(text+i) = 'a';
         int m,n;
           n=1000;
           m=10;
      while(m<=1000)
    {
        pattern = (char *)malloc(m*sizeof(char));
       //For Best case
            for(int i=0; i<m; i++)
                pattern[i] = 'a';
```

```
        count=0;
      stringmatching(text, pattern, n,m);
        fprintf(f1, "%d\t%d\n", m, count); //printf("%d\t%d\n", m, count);
          count = 0;
     //For Worst case
          count=0;
          pattern[m-1] = 'b';
      stringmatching(text, pattern, n,m);
       fprintf(f2, "%d\t%d\n", m, count); //printf("%d\t%d\n", m, count);
     //For Average Case
       for(int i=0; i<m; i++)
       pattern[i] =97+ rand()%3;
        count=0;
        stringmatching(text, pattern, n,m);
        fprintf(f3,"%d\t%d\n", m, count); //printf("%d\t%d\n", m, count);
          // for(int i=0; i<m; i++)
          //  printf("%c ",pattern[i]);
           //  printf("\n");
              count=0;
              free(pattern);
       if(m<100)
        m=m+10;
        else
         m=m+100;
 }
}
void main()
```

{

ploter(); }

OUTPUT

TESTER

```
ENTER THE PATTERN LENGTH
10
ENTER THE PATTERN
Hey there
ENTER THE TEXT LENGTH
30
ENTER THE TEXT
Hello there , How are you doing
THE PATTERN not found
Number of comparisons: 24
```

PLOTTER

Best case

Open ▾ stringbest.txt ~/ADA

```
10      10
20      20
30      30
40      40
50      50
60      60
70      70
80      80
90      90
100     100
200     200
300     300
400     400
500     500
600     600
700     700
800     800
900     900
1000    1000
```

Worst case

Open ▾ stringworst.txt ~/ADA

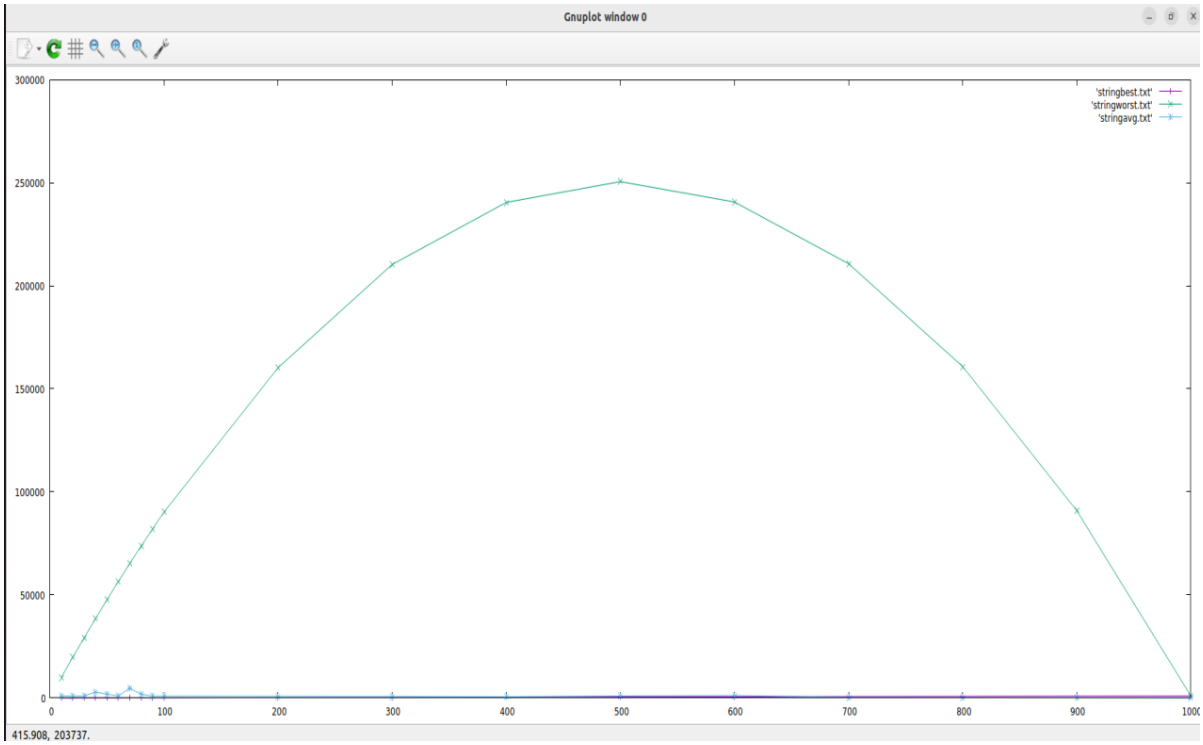
```
10      9910
20      19620
30      29130
40      38440
50      47550
60      56460
70      65170
80      73680
90      81990
100     90100
200     160200
300     210300
400     240400
500     250500
600     240600
700     210700
800     160800
900     90900
1000    1000
```

Average case

stringavg.txt
~/ADA

```
10      991
20      981
30      971
40      2883
50      1902
60      941
70      4655
80      1842
90      911
100     901
200     801
300     701
400     601
500     1002
600     1203
700     301
800     201
900     101
1000    3
```

Gnuplot



5.Implement the Merge sort algorithm and perform its analysis for worst case, best case and average inputs.

Tester code:

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

int count;

```
void merge(int *arr,int beg,int mid,int end)
{
  int i,j,k;
  int n1=(mid-beg)+1;
  int n2=end-mid;
  int left[n1],right[n2];
  for(i=0;i<n1;i++)
  left[i]=arr[beg+i];
  for(j=0;j<n2;j++)
  right[j]=arr[mid+j+1];
   i=0;j=0;k=beg;
  while(i<n1&&j<n2)
  {
      count++;
     if(left[i]<=right[j])
      arr[k]=left[i++];
     else
      arr[k]=right[j++];
     k++;
  }
   while(i<n1)
   arr[k++]=left[i++];
    while(j<n2)
   arr[k++]=right[j++];

}
void mergesort(int *arr,int beg,int end)
```

```
{
   if(beg<end)
   {
    int mid=(beg+end)/2;
    mergesort(arr,beg,mid);
    mergesort(arr,mid+1,end);
    merge(arr,beg,mid,end);
   }
}
void main()
{
  int *arr, n;
   printf("ENTER THE NUMBER OF ELEMENTS\n");
   scanf("%d",&n);

   arr=(int *)malloc(sizeof(int)*n);
   printf("ENTER THE ELEMENTS OF THE ARRAY\n");
      for(int i=0;i<n;i++)
       scanf("%d",&arr[i]);

  printf("THE ELEMENTS OF THE ARRAY BEFORE SORTING\n");
    for(int i=0;i<n;i++)
       printf("%d ",arr[i]);
    printf("\n");

      mergesort(arr,0,n-1);
```

```
    printf("THE ELEMENTS OF THE ARRAY BEFORE SORTING\n");
   for(int i=0;i<n;i++)
     printf("%d ",arr[i]);
     printf("\n");
     printf("\n");
}
```

OUTPUT:

```
ENTER THE NUMBER OF ELEMENTS
5
ENTER THE ELEMENTS OF THE ARRAY
0
-5
9
1
-2
THE ELEMENTS OF THE ARRAY BEFORE SORTING
0 -5 9 1 -2
THE ELEMENTS OF THE ARRAY BEFORE SORTING
-5 -2 0 1 9
```

PLOTTER code;

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
int count;
void merge(int *arr,int beg,int mid,int end)
{
  int i,j,k;
  int n1=(mid-beg)+1;
  int n2=end-mid;
  int left[n1],right[n2];
```

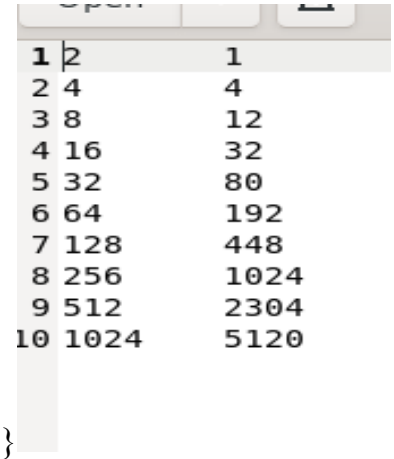
```
   for(i=0;i<n1;i++)
   left[i]=arr[beg+i];
   for(j=0;j<n2;j++)
   right[j]=arr[mid+j+1];
    i=0;j=0;k=beg;
   while(i<n1&&j<n2)
   {
       count++;
      if(left[i]<=right[j])
       arr[k]=left[i++];
      else
       arr[k]=right[j++];
      k++;
   }

    while(i<n1)
    arr[k++]=left[i++];
     while(j<n2)
    arr[k++]=right[j++];

}



void mergesort(int *arr,int beg,int end)
{
    if(beg<end)
```

```
    {
     int mid=(beg+end)/2;
     mergesort(arr,beg,mid);
     mergesort(arr,mid+1,end);
     merge(arr,beg,mid,end);
    }
}


void worst(int arr[],int beg,int end)
{
  if(beg<end)
  {
  int mid=(beg+end)/2;
  int i,j,k;
  int n1=(mid-beg)+1;
  int n2=end-mid;
  int a[n1],b[n2];
  for(i=0;i<n1;i++)
  a[i]=arr[(2*i)];
  for(j=0;j<n2;j++)
  b[j]=arr[(2*j)+1];

  worst(a,beg,mid);
  worst(b,mid+1,end);

 for(i=0;i<n1;i++)
```

```
  arr[i]=a[i];
  for(j=0;j<n2;j++)
  arr[j+i]=b[j];

 }

}

void main()
{
  int *arr,n;
  srand(time(NULL));
  FILE *f1,*f2,*f3,*f4;
  f1=fopen("MERGESORTBEST.txt","a");
  f2=fopen("MERGESORTWORST.txt","a");
  f3=fopen("MERGESORTAVG.txt","a");
  f4=fopen("WORSTDATA.txt","a");

   for(n=2;n<=1024;n=n*2)
   {
     arr=(int *)malloc(sizeof(int)*n);
    for(int i=0;i<n;i++)
    *(arr+i)=i+1;
     count=0;
    //Best case
    mergesort(arr,0,n-1);
      fprintf(f1,"%d\t%d\n",n,count);
```

```
    //worst case
     count=0;
     worst(arr,0,n-1);
      for(int i=0;i<n;i++)
     fprintf(f4,"%d ",*(arr+i));
     fprintf(f4,"\n");
     mergesort(arr,0,n-1);
     fprintf(f2,"%d\t%d\n",n,count);

    //AVG case
     for(int i=0;i<n;i++)
     *(arr+i)=rand()%n;
     count=0;
     mergesort(arr,0,n-1);
     fprintf(f3,"%d\t%d\n",n,count);
     free(arr);
   }
   fclose(f1);
   fclose(f2);
   fclose(f3);
   fclose(f4);

  printf("DATA IS ENTERED IN TO FILE\n");
}
```

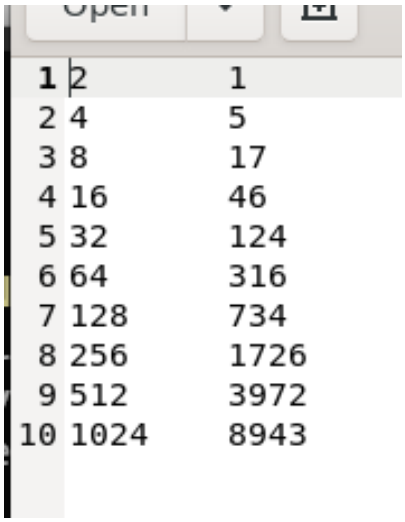
Best case :



1	2	1
2	4	4
3	8	12
4	16	32
5	32	80
6	64	192
7	128	448
8	256	1024
9	512	2304
10	1024	5120

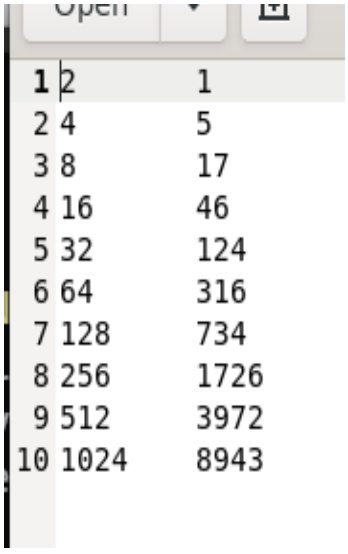
}

Worst case:



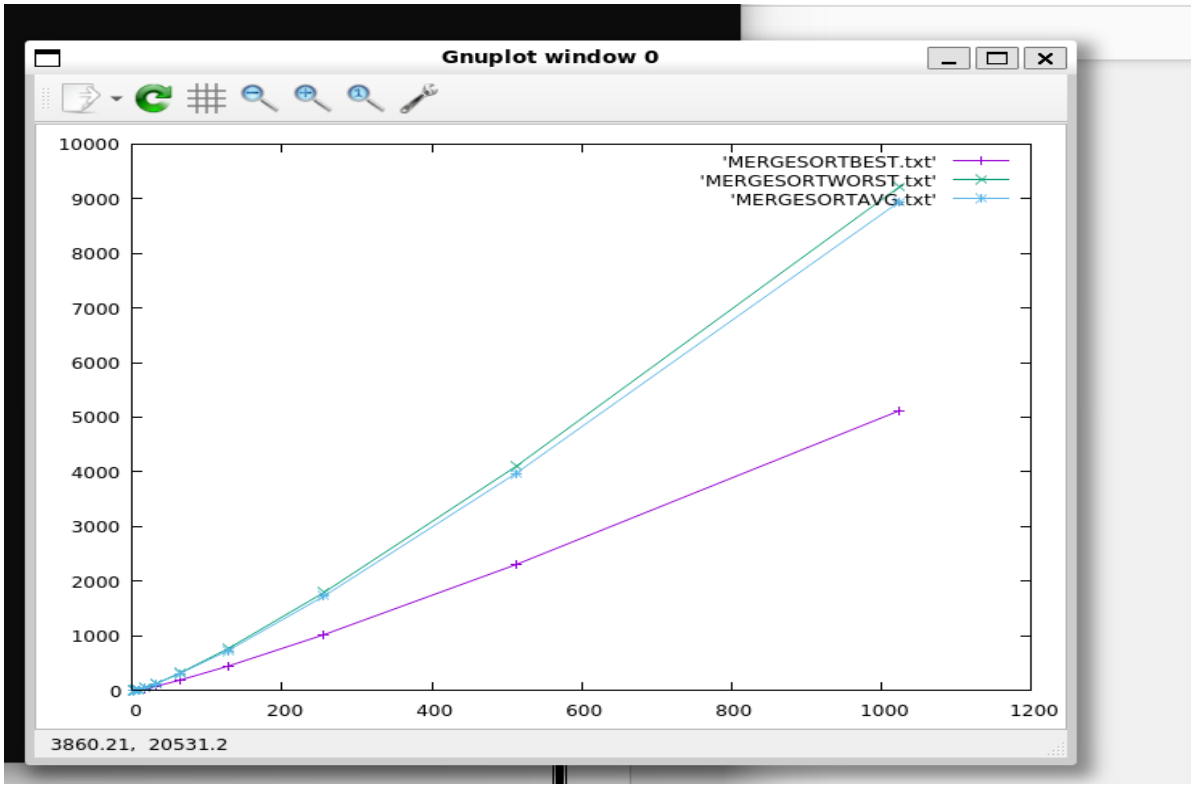
1	2	1
2	4	5
3	8	17
4	16	46
5	32	124
6	64	316
7	128	734
8	256	1726
9	512	3972
10	1024	8943

Average case



1	2	1
2	4	5
3	8	17
4	16	46
5	32	124
6	64	316
7	128	734
8	256	1726
9	512	3972
10	1024	8943

Gnuplot :



6.Implement Quick sort algorithm and perform its analysis for worst case, best case and average inputs.

PLOTTER

```
/* program to implement quick sort*/
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
  int count;
void swap(int  *a,int *  b)
{
     int temp=*a;
      *a=*b;
```

```
      *b=temp;
}
int partition(int * arr,int beg,int end)
{
  int pivot =arr[beg];
  int i=beg,j=end+1;
 do{
   do{
       count++;
        i++;
       }while(arr[i]<pivot);
  do{
       count++;
       j--;
     }while(arr[j]>pivot);
     swap(&arr[i],&arr[j]);
  }while(i<j);
  swap(&arr[i],&arr[j]);
  swap(&arr[beg],&arr[j]);
return j;
}
void quicksort(int *arr,int beg,int end)
{
 if(beg<end)
{
   int split=partition(arr,beg,end);
   quicksort(arr,beg,split-1);
```

```
    quicksort(arr,split+1,end);
 }
}
void main()
{
   int *arr,n;
    srand(time(NULL));
    FILE *f1,*f2,*f3;
     f1=fopen("QUICKBEST.txt","a");
     f2=fopen("QUICKWORST.txt","a");
     f3=fopen("QUICKAVG.txt","a");
 n=4;
while(n<1034)
{
    arr=(int *)malloc(sizeof(int)*n);
    for(int i=0;i<n;i++)
         *(arr+i)=5;
          count=0;
  //Best case
   quicksort(arr,0,n-1);
   fprintf(f1,"%d\t%d\n",n,count);  //printf("%d\t%d\n",n,count);
    //worst case
    count=0;
    for(int i=0;i<n;i++)
         *(arr+i)=i+1;
          quicksort(arr,0,n-1);
     fprintf(f2,"%d\t%d\n",n,count); //printf("%d\t%d\n",n,count);
```

```
 //AVG case
      for(int i=0;i<n;i++)
       *(arr+i)=rand()%n;
          count=0;
       quicksort(arr,0,n-1);
      fprintf(f3,"%d\t%d\n",n,count); //printf("%d\t%d\n",n,count);
        n=n*2;
        free(arr);
 }
    fclose(f1);
    fclose(f2);
    fclose(f3);
}
```

TESTER

```
/* program to implement quick sort*/
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
   int count;
void swap(int  *a,int *  b)
{
      int temp=*a;
       *a=*b;
       *b=temp;
}
int partition(int * arr,int beg,int end)
{
```

```
  int pivot =arr[beg];
  int i=beg,j=end+1;
 do{
   do{
       count++;
        i++;
       }while(arr[i]<pivot);
 do{
       count++;
       j--;
     }while(arr[j]>pivot);
     swap(&arr[i],&arr[j]);
  }while(i<j);
  swap(&arr[i],&arr[j]);
  swap(&arr[beg],&arr[j]);
return j;
}
void quicksort(int *arr,int beg,int end)
{
 if(beg<end)
{
   int split=partition(arr,beg,end);
   quicksort(arr,beg,split-1);
    quicksort(arr,split+1,end);
 }
}
void main()
```

```
{
   int *arr, n;
    printf("ENTER THE NUMBER OF ELEMENTS\n");
     scanf("%d",&n);
arr=(int *)malloc(sizeof(int)*n);
printf("ENTER THE ELEMENTS OF THE ARRAY\n");
 for(int i=0;i<n;i++)
    scanf("%d",&arr[i]);
printf("THE ELEMENTS OF THE ARRAY BEFORE SORTING\n");
for(int i=0;i<n;i++)
printf("%d ",arr[i]);
printf("\n");
quicksort(arr,0,n-1);
 printf("THE ELEMENTS OF THE ARRAY BEFORE SORTING\n");
for(int i=0;i<n;i++)
 printf("%d ",arr[i]);
   printf("\n");
   printf("\n");
}
```

OUTPUT

PLOTTER

Best case

Worst case

Open ▾ QUICKWORST.txt ~/ADA

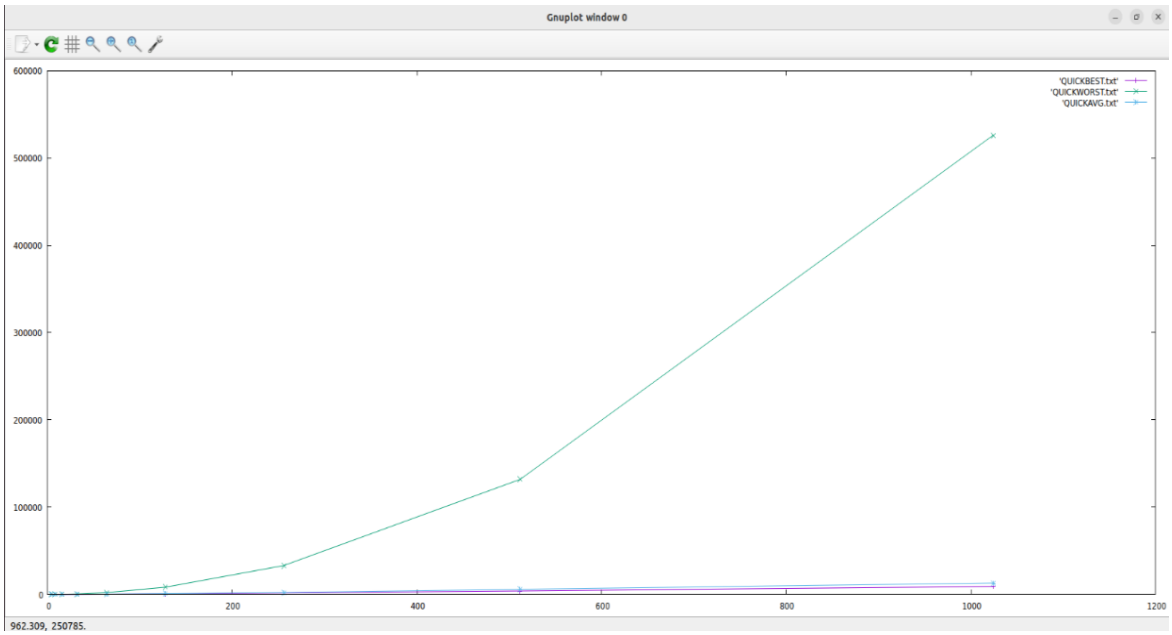
```
4       12
8       42
16      150
32      558
64      2142
128     8382
256     33150
512     131838
1024    525822
```

Average case

Open ▾ QUICKAVG.txt ~/ADA

```
4       8
8       20
16      73
32      170
64      441
128     1059
256     2265
512     5975
1024    13201
```

Gnuplot



TESTER

```
ENTER THE NUMBER OF ELEMENTS
10
ENTER THE ELEMENTS OF THE ARRAY
9 5 8 2 0 -1 8 4 1 0
THE ELEMENTS OF THE ARRAY BEFORE SORTING
9 5 8 2 0 -1 8 4 1 0
THE ELEMENTS OF THE ARRAY BEFORE SORTING
-1 0 0 1 2 4 5 8 8 9
```

7.Implement DFS algorithm to check for connectivity and acyclicity of a graph. If not connected, display the connected components. Perform its analysis by generating best case and worst case data. Note: while showing correctness, input should be given for both connected/disconnected and cyclic/acyclic graphs.

Adjacency list:

TESTER:

```
#include<stdio.h>
#include<stdlib.h>


 struct node
{
  int info;
  struct node *next;
};



struct Graph{
  int vertices;
  int edges;
   int * visit;
   struct node ** adjLists;
};
```

```
typedef struct node * Node;

Node createnode(int n)
{
  Node nn=(Node)malloc(sizeof(struct node));
  nn->info=n;
  nn->next=NULL;
 return nn;
}

 struct Graph* createGraph(int vertices)
{
struct Graph* graph = malloc(sizeof(struct Graph));
graph->vertices = vertices;

graph->adjLists = malloc(vertices * sizeof(struct node*));

graph->visit = malloc(vertices * sizeof(int));

int i;
for (i = 0; i < vertices; i++) {
  graph->adjLists[i]= NULL;
  graph->visit[i] = 0;
}
return graph;
}
```

```
int count=0,iscyclic=0;

  void DFS(struct Graph* graph, int vertex,int parent)
 {
 struct node* adjList = graph->adjLists[vertex];
 struct node* temp = adjList;

 count++;
 graph->visit[vertex] = 1;
 printf("-->%c ", vertex+65);

 while (temp != NULL)
 {
  int connectedVertex = temp->info;
  if (graph->visit[connectedVertex]==1&&connectedVertex!=parent)
      {
          iscyclic=1;
      }
  if (graph->visit[connectedVertex] == 0)
       DFS(graph,connectedVertex,vertex);

  temp = temp->next;
 }
}

void main()
```

```
{
   int n;
    printf("ENTER THE NUUMBER OF VERTICES\n");
    scanf("%d",&n);
    struct Graph*   g=createGraph(n);
    Node temp;
    int key;


printf("Enter the adjacency LIST \n");
 for(int i=0;i<g->vertices;i++)
{
 printf("Enter 1 for the vertices adjacent to vertex %c\n",i+65);
  for(int j=0;j<g->vertices;j++)
 {

   if(i!=g->vertices-j-1)
     {
         Node nn=createnode(g->vertices-j-1);
         nn->next = g->adjLists[i];
         g->adjLists[i] = nn;
     }
  /* 11:  printf("\nVertex %c : ",g->vertices-j-1+65);
     scanf("%d",&key);
     if()
     {
         Node nn=createnode(g->vertices-j-1);
```

```
            nn->next = g->adjLists[i];
            g->adjLists[i] = nn;
        }
        else if(key!=0)
        {
         printf("Enter 1 to add and 0 to not \n");
        goto l1;
        }*/
    }
}

     printf("\n");
    for(int i=0;i<g->vertices;i++)
    {
      temp=g->adjLists[i];
      printf("THE VERTEX ADJACENT TO %c : ",i+65);
       while(temp!=NULL)
        {
         printf("%c ",temp->info+65);
         temp=temp->next;
        }
      printf("\n");
    }
   int dfscount=0;
    printf("\nDFS TRAVERSAL STARTING FROM NODE %C\n",65);
    DFS(g,0,-1);
     dfscount++;
```

```
  if(count==g->vertices)
   printf("\n THE GRAPH IS CONNECTED\n");
   else
   {
    printf("\nTHE GRAPH IS NOT CONNECTED\n");
    int start=1;
     while(count!=g->vertices)
       {
         if(g->visit[start]!=1)
          {
            printf("\n");
            DFS(g,start,-1);
            dfscount++;
          }
         start++;
       }

   }

   if(iscyclic==1)
   {
     printf("\nTHE GRAPH  HAS A  CYCLE \n");
   }
   else
   printf("\nTHE GRAPH  DONT HAVE A  CYCLE \n");
  }
/*program to implement dfs with adjacency list with graph */
```

```
#include<stdio.h>
#include<stdlib.h>
int gcount=0;
        struct node
         {
         int info;
         struct node *next;
         };
         struct Graph{
         int vertices;
         int edges;
         int * visit;
         struct node ** adjLists;
         };

         typedef struct node * Node;

         Node createnode(int n)
         {
         Node nn=(Node)malloc(sizeof(struct node));
         nn->info=n;
         nn->next=NULL;
         return nn;
         }

         struct Graph* createGraph(int vertices)
```

```
{
struct Graph* graph = malloc(sizeof(struct Graph));
graph->vertices = vertices;

graph->adjLists = malloc(vertices * sizeof(struct node*));

graph->visit = malloc(vertices * sizeof(int));

int i;
for (i = 0; i < vertices; i++) {
graph->adjLists[i]= NULL;
graph->visit[i] = 0;
}
return graph;
}

int count=0,iscyclic=0;

void DFS(struct Graph* graph, int vertex,int parent)
{
struct node* adjList = graph->adjLists[vertex];
struct node* temp = adjList;

count++;
graph->visit[vertex] = 1;
printf("-->%c ", vertex+65);
while (temp != NULL)
```

```
  {
   gcount++;
  int connectedVertex = temp->info;
  if (graph->visit[connectedVertex]==1&&connectedVertex!=parent)
  {
  iscyclic=1;
  }
  if (graph->visit[connectedVertex] == 0)
  DFS(graph,connectedVertex,vertex);

  temp = temp->next;
  }
   gcount++;

  }
   void ploter(int k)
  {
  FILE *fp1=fopen("dfsbest.txt","a");
  FILE *fp2=fopen("dfsworst.txt","a");

 for(int i=1;i<=10;i++)
{
  int n;
 n=i;
  struct Graph* g=createGraph(n);
  Node temp;
  int key;
```

```
 if(k==0)
{
 for(int i=0;i<g->vertices-1;i++)
{

  Node nn=createnode(i+1);
      nn->next = g->adjLists[i];
     g->adjLists[i] = nn;
}
  }

if(k==1)
{
 for(int i=0;i<g->vertices;i++)
 {
  for(int j=0;j<g->vertices;j++)
  {
   if(i!=g->vertices-j-1)
   {
    Node nn=createnode(g->vertices-j-1);
     nn->next = g->adjLists[i];
    g->adjLists[i] = nn;
   }
  }
 }
```

```
  }
  printf("\n");
   for(int i=0;i<g->vertices;i++)
  {
  temp=g->adjLists[i];
  printf("THE VERTEX ADJACENT TO %c : ",i+65);
  while(temp!=NULL)
  {
  printf("%c ",temp->info+65);
  temp=temp->next;
  }
  printf("\n");
  }
  gcount=0;
iscyclic=0;
count=0;
  int dfscount=0;
  printf("\nDFS TRAVERSAL STARTING FROM NODE %C\n",65);
  DFS(g,0,-1);
  dfscount++;
  if(count==g->vertices)
  printf("\n THE GRAPH IS CONNECTED\n");
  else
  {
  printf("\nTHE GRAPH IS NOT CONNECTED\n");
  int start=1;
  while(count!=g->vertices)
```

```
        {
        if(g->visit[start]!=1)
        {
        printf("\n");
        DFS(g,start,-1);
       }
      start++;
    }
  }
   if(iscyclic)
   printf("THE GRAPH HAS A CYCLE\n");
   else
   {
     printf("THE GRAPH donot have A CYCLE\n");
   }
   if(k==0)
    fprintf(fp1,"%d\t%d\n",n,gcount);
   else
    fprintf(fp2,"%d\t%d\n",n,gcount);
  }
}

void main()
{
   for(int i=0;i<2;i++)
    ploter(i);
}
```

ADJACENCY MATRIX:

```
#include<stdio.h>
#include<stdlib.h>

int graph[100][100], visited[100], isCyclic = 0;
int dfsCount = 0, count = 0;
int dcount=0;
int path[100];
int d;

void dfs1(int n, int start, int parent) {
   visited[start] = 1;
   count++;
   for(int i=0; i<n; i++) {
      if(i!=parent && graph[start][i] && visited[i])
         isCyclic = 1;

         dcount++;
      if(graph[start][i] && visited[i]==0)
         dfs1(n, i, start);
   }
}

void ploter(int k)
{

   FILE *f1= fopen("DFSBEST.txt", "a");
```

```
   FILE *f2=fopen("DFSWORsT.txt", "a");
   int v;
   for(int i=1;i<=10;i++)
  {
   v=i;



 if(k==0)
 {
  for(int i=0;i<v;i++)
{

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

      if(i!=j)
      {
       graph[i][j] =1;
      }
      else
      graph[i][j] =0;
  }
}
 }

if(k==1)
```

```
{
    for(int i=0;i<v;i++)
    {
        for(int j=0;j<v;j++)
         graph[i][j] =0;
    }
    for(int i=0;i<v-1;i++)
    {
            graph[i][i+1]=1;
    }

}
    isCyclic=0;
     dfsCount = 0;
     count = 0;
     dcount=0;
    dfs1(v, 0, -1);
    dfsCount++;
    int start;
        start = 1;
        while(count != v) {
            if(visited[start] != 1) {
                dfs1(v, start, -1);
                dfsCount++;
            }
            start++;
        }
```

```
        if(k==0)
       fprintf(f2,"%d\t%d\n",v,dcount);
       else
        fprintf(f1,"%d\t%d\n",v,dcount);

  }

  fclose(f1);
  fclose(f2);
}
void main()
{
      for(int i=0;i<2;i++)
            ploter(i);
         printf("DATA ENTERED IN TO THE FILE\n");
}
```

PLOTTER:

/*program to implement the dfs algorithm and to check connectivity and acyclicity with adjacency matrix */

```
#include<stdio.h>
#include<stdlib.h>

int graph[100][100], visited[100], isCyclic = 0;
int dfsCount = 0, count = 0;
int dcount=0;
int path[100];
```

```
int d;
void dfs(int n, int start, int parent) {
    visited[start] = 1;
     path [start]=1;
    count++;
    printf("--> %c ", start+65);
    for(int i=0; i<n; i++) {

       if(d==1)
       {
         if(i!=parent && graph[start][i] && visited[i]==1 && path[i]==1)
          isCyclic = 1;
       }
         else
         {
       if(i!=parent && graph[start][i] && visited[i])
          isCyclic = 1;
         }
          dcount++;
       if(graph[start][i] && visited[i]==0)
          dfs(n, i, start);
    }
     path [start]=0;
}


void main(){
```

```
int n, start;
dfsCount = 0;
 count = 0;
  dcount=0;
  d=0;
printf("Enter the number of nodes in the graph:\n");
scanf("%d", &n);
printf("Enter the Adjacency Matrix:\n");
for(int i=0; i<n; i++){
    for(int j=0; j<n; j++){
        scanf("%d", &graph[i][j]);
    }
    visited[i] = 0;
    path[i] =0;
}
printf("enter is the 1 graph is directed to:\n");
scanf("%d", &d);

printf("the Adjacency Matrix:\n");
for(int i=0; i<n; i++){
    for(int j=0; j<n; j++){
        printf("%d ", graph[i][j]);
    }
    printf("\n");
}
isCyclic =0;
printf("\nDFS traversal starting from node %c\n", 65);
```

```
    dfs(n, 0, -1);
    dfsCount++;
    if(count == n)
       printf("\nThe Graph is connected\n");
    else {
       printf("\nThe Graph is not connected\n");
       start = 1;
       while(count != n) {
          if(visited[start] != 1) {
             printf("\n");
             dfs(n, start, -1);
             dfsCount++;
          }
          start++;
       }
    }
    printf("\nThe number of components is %d\n", dfsCount);

    if(isCyclic)
       printf("\nThe graph is cyclic\n");
    else
       printf("\nThe graph is not cyclic\n");
}
```

OUTPUT:

```
enter is the 1 graph is directed to:
1
the Adjacency Matrix:
0 0 1 0 0
0 0 1 0 0
0 0 0 1 1
0 0 0 0 1
0 0 0 0 0

DFS traversal starting from node A
--> A --> C --> D --> E
The Graph is not connected

--> B
The number of components is 2

The graph is not cyclic
```

```
enter is the 1 graph is directed to:
0
the Adjacency Matrix:
0 1 1
1 0 1
1 1 0

DFS traversal starting from node A
--> A --> B --> C
The Graph is connected

The number of components is 1

The graph is cyclic
```

LINKED LIST:

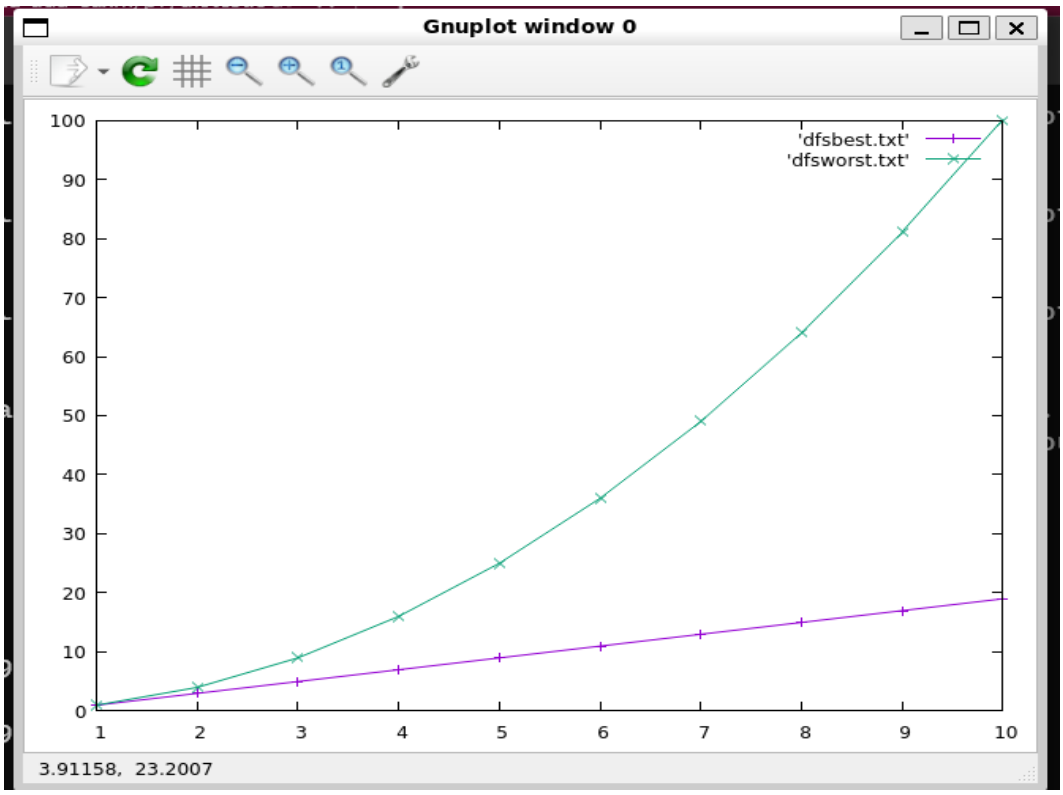
Best case:

1	1	1
2	2	3
3	3	5
4	4	7
5	5	9
6	6	11
7	7	13
8	8	15
9	9	17
10	10	19

Worst case:

1	1	1
2	2	4
3	3	9
4	4	16
5	5	25
6	6	36
7	7	49
8	8	64
9	9	81
10	10	100
11		

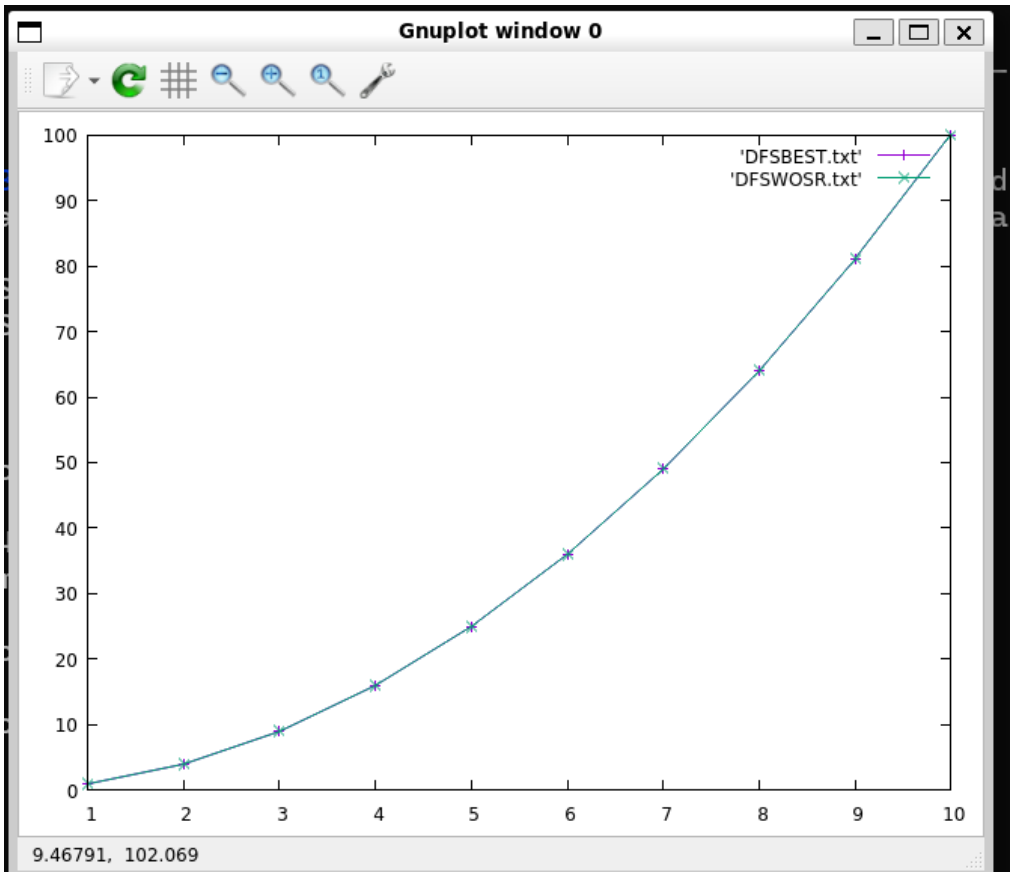
Graph:



WITH ADJACENCY MATRIX:

```
 1 1         1
 2 2         4
 3 3         9
 4 4         16
 5 5         25
 6 6         36
 7 7         49
 8 8         64
 9 9         81
10 10        100
11
```

Graph:



8.Implement BFS algorithm to check for connectivity and acyclicity of a graph. If not connected, display the connected components. Perform its analysis by generating best case and worst case data. Note: while showing correctness, input should be given for both connected/disconnected and cyclic/acyclic graphs.

Adjacency list:

```
#include<stdio.h>
#include<stdlib.h>


 struct node
{
   int info;
   struct node *next;
};
struct Graph{
   int vertices;
   int edges;
    int * visit;
    struct node ** adjLists;
};
  typedef struct node * Node;
  Node createnode(int n)
  {
     Node nn=(Node)malloc(sizeof(struct node));
     nn->info=n;
     nn->next=NULL;
    return nn;
  }
```

```
struct Graph* createGraph(int vertices) {
 struct Graph* graph = malloc(sizeof(struct Graph));
 graph->vertices = vertices;

 graph->adjLists = malloc(vertices * sizeof(struct node*));

 graph->visit = malloc(vertices * sizeof(int));

 int i;
 for (i = 0; i < vertices; i++) {
  graph->adjLists[i]= NULL;
  graph->visit[i] = 0;
 }
 return graph;
}

int count=0,iscyclic=0;
int ordercount=0;
void bfs(struct Graph* graph, int start)
{
   int queue[15], parent[15];
   int rear = -1, front = -1, i, parentNode;
   graph->visit[start] = 1;
   count++;
   queue[++rear] = start;
   parent[rear] = -1;
   while(rear != front){
```

```
        start = queue[++front];
        parentNode = parent[front];
        printf("-->%c", start+65);
         Node temp = graph->adjLists[start];
        while (temp !=NULL){
        {
           int connectedVertex = temp->info;
           ordercount++;
           if (connectedVertex != parentNode&& graph->visit[connectedVertex])
              iscyclic = 1;
           if(graph->visit[connectedVertex]==0){
              queue[++rear] = connectedVertex;
              parent[rear] = start;
              graph->visit[connectedVertex] = 1;
              count++;
           }
            temp = temp->next;
        }
    }
}
}

void main()
{
    int n;
     printf("ENTER THE NUUMBER OF VERTICES\n");
    scanf("%d",&n);
```

```
 struct Graph*   g=createGraph(n);
 Node temp;
int key;


printf("Enter the adjacency LIST \n");
 for(int i=0;i<g->vertices;i++)
{
 printf("Enter 1 for the vertices adjacent to vertex %c\n",i+65);
  for(int j=0;j<g->vertices;j++)
 {
    printf("\nVertex %c : ",j+65);
    scanf("%d",&key);
         Node nn=createnode(j);
         nn->next = g->adjLists[i];
         g->adjLists[i] = nn;

    /* this is for the file and data generation
     if(i!=g->vertices-1-j)
     {
         Node nn=createnode(j);
         nn->next = g->adjLists[i];
         g->adjLists[i] = nn;

     }*/
 }
}
```

```
 for(int i=0;i<g->vertices;i++)
{
   temp=g->adjLists[i];
   printf("THE VERTEX ADJACENT TO %c : ",i+65);
    while(temp!=NULL)
     {
      printf("%c ",temp->info+65);
      temp=temp->next;
     }
   printf("\n");
 }
int bfscount=0;
 printf("BFS TRAVERSAL STARTING FROM NODE %C\n",65);
 bfs(g,0);
  bfscount++;
 if(count==g->vertices)
  printf("the graph is connected \n");
  else
  {
  printf("\nthe graph is not connected ");
      int start=1;
     while(count!=g->vertices)
       {
          if(g->visit[start]!=1)
           {
```

```
            printf("\n");
            bfs(g,start);
            bfscount++;
          }
          start++;
        }

    }


    printf("\nTHE NUMBER OF CONNECTED COMPONENTS ARE
%d\n",bfscount);
    if(iscyclic)
    printf("the graph is cyclic\n");
    else
    printf("the graph is not cyclic\n");
}
```

Adjacency matrix:

```
#include<stdio.h>
#include<stdlib.h>
int bfsCount = 0, cyclic=0;
int count = 0;//to count how many vertex visited
int orderCount = 0;

int graph[100][100], visited[100];

void bfs(int n, int start){
    int queue[n], parent[n];
```

```
    int rear = -1, front = -1, i, parentNode;
    visited[start] = 1; count++;
    queue[++rear] = start;
    parent[rear] = -1;
    while(rear != front){
        start = queue[++front];
        parentNode = parent[front];
        printf("-->%c", start+65);
        for(i=0; i<n; i++){
            orderCount++;
            if (i != parentNode && graph[start][i] && visited[i])
                cyclic = 1;
            if((graph[start][i]) && (visited[i] == 0)){
                queue[++rear] = i;
                parent[rear] = start;
                visited[i] = 1;
                count++;
            }
        }
    }
}


void bfs1(int n, int start){
    int queue[n], parent[n];
    int rear = -1, front = -1, i, parentNode;
    visited[start] = 1; count++;
```

```
    queue[++rear] = start;
    parent[rear] = -1;
    while(rear != front){
        start = queue[++front];
        parentNode = parent[front];
        for(i=0; i<n; i++){
            orderCount++;
            if (i != parentNode && graph[start][i] && visited[i])
                cyclic = 1;
            if((graph[start][i]) && (visited[i] == 0)){
                queue[++rear] = i;
                parent[rear] = start;
                visited[i] = 1;
               // count++;
            }
        }
    }
}
void tester(){
    int n, i, j, start;
    printf("Enter the number of nodes in the graph:\n");
    scanf("%d", &n);
    printf("Enter the Adjacency Matrix:\n");
    for(i=0; i<n; i++){
        for(j=0; j<n; j++){
            scanf("%d", &graph[i][j]);
        }
```

```
        visited[i] = 0;
    }

     printf("the Adjacency Matrix:\n");
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            printf("%d ", graph[i][j]);
        }
        printf("\n");
    }
     bfsCount = 0, cyclic=0;
count = 0;
 orderCount = 0;
    printf("Breadth First Search Traversal:\n");
    bfsCount++;
    bfs(n, 0);
    if(count == n){
        printf("\nGraph is connected.\n");
    }
    else {
        printf("\nThe graph is not connected.\n");
        start = 1;
        while(count != n){
            if(visited[start] != 1) {
                bfsCount++;
                bfs(n, start);
                printf("\n");
```

```
            }
            start++;
        }
    }
    printf("\nThe number of components in the graph is %d\n", bfsCount);
    if(cyclic) {
        printf("\nThe graph is cyclic\n");
    } else {
        printf("\nThe graph is acyclic\n");
    }
}



void ploter(int k)
{

    FILE *f1= fopen("BFSBEST.txt", "a");
    FILE *f2=fopen("BFSWOSR.txt", "a");
    int v,start;
    for(int i=1;i<=10;i++)
  {
    v=i;
  int *arr[v];
  for(int i=0;i<v;i++)
  arr[i]=(int *)malloc(sizeof(int)*v);
```

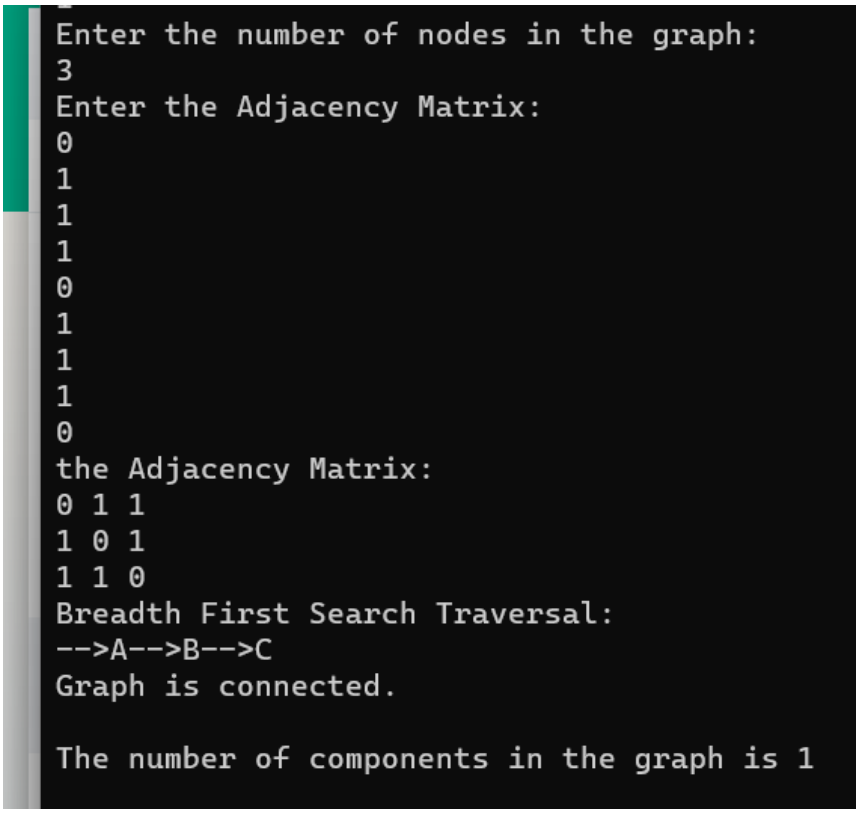
```
if(k==0)
 {
  for(int i=0;i<v;i++)
{

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

      if(i!=j)
      {
       arr[i][j] =1;
      }
      else
      arr[i][j] =0;
  }
}
 }

if(k==1)
{
    for(int i=0;i<v;i++)
    {
        for(int j=0;j<v;j++)
         arr[i][j] =0;
    }
    for(int i=0;i<v-1;i++)
```

```
    {
        arr[i][i+1]=1;
    }

}

    bfsCount = 0, cyclic=0;
    count = 0;
    orderCount = 0;
    bfsCount++;
    bfs1(v, 0);
    if(count != v){
       start = 1;
       while(count != v){
          if(visited[start] != 1) {
             bfsCount++;
             bfs1(v, start);
          }
          start++;
       }
    }
         if(k==0)
        fprintf(f2,"%d\t%d\n",v,orderCount);
        else
         fprintf(f1,"%d\t%d\n",v,orderCount);
        // printf("%d\t%d\n",v,orderCount);
```

```
  }

  fclose(f1);
  fclose(f2);

}



void main()
{
   for(;;)
   {
      int key;
      printf("ENTER THE CHOICE 1.TO TEST \n2.TO PLOT\nOTHER TO
EXIT\n");
      scanf("%d",&key);


       switch(key)
       {
         case 1:tester();break;
         case 2:for(int i=0;i<2;i++)
                ploter(i);
                break;
         default:exit(1);
       }

   }
```

}

OUTPUT :


Enter the number of nodes in the graph:
3
Enter the Adjacency Matrix:
0
1
1
1
0
1
1
1
0
the Adjacency Matrix:
0 1 1
1 0 1
1 1 0
Breadth First Search Traversal:
-->A-->B-->C
Graph is connected.
The number of components in the graph is 1

```
OTHER TO EXIT
1
Enter the number of nodes in the graph:
3
Enter the Adjacency Matrix:
0
1
0
0
0
1
0
0
0
the Adjacency Matrix:
0 1 0
0 0 1
0 0 0
Breadth First Search Traversal:
-->A-->B-->C
Graph is connected.

The number of components in the graph is 1

The graph is acyclic
```

WITH LINKED LIST:

```
vertex C : 0
THE VERTEX ADJACENT TO A : C B A
THE VERTEX ADJACENT TO B : C B A
THE VERTEX ADJACENT TO C : C B A
BFS TRAVERSAL STARTING FROM NODE A
-->A-->C-->Bthe graph is connected

THE NUMBER OF CONNECTED COMPONENTS ARE 1
the graph is cyclic
hp@LAPTOP-28MPKT32:~$
```

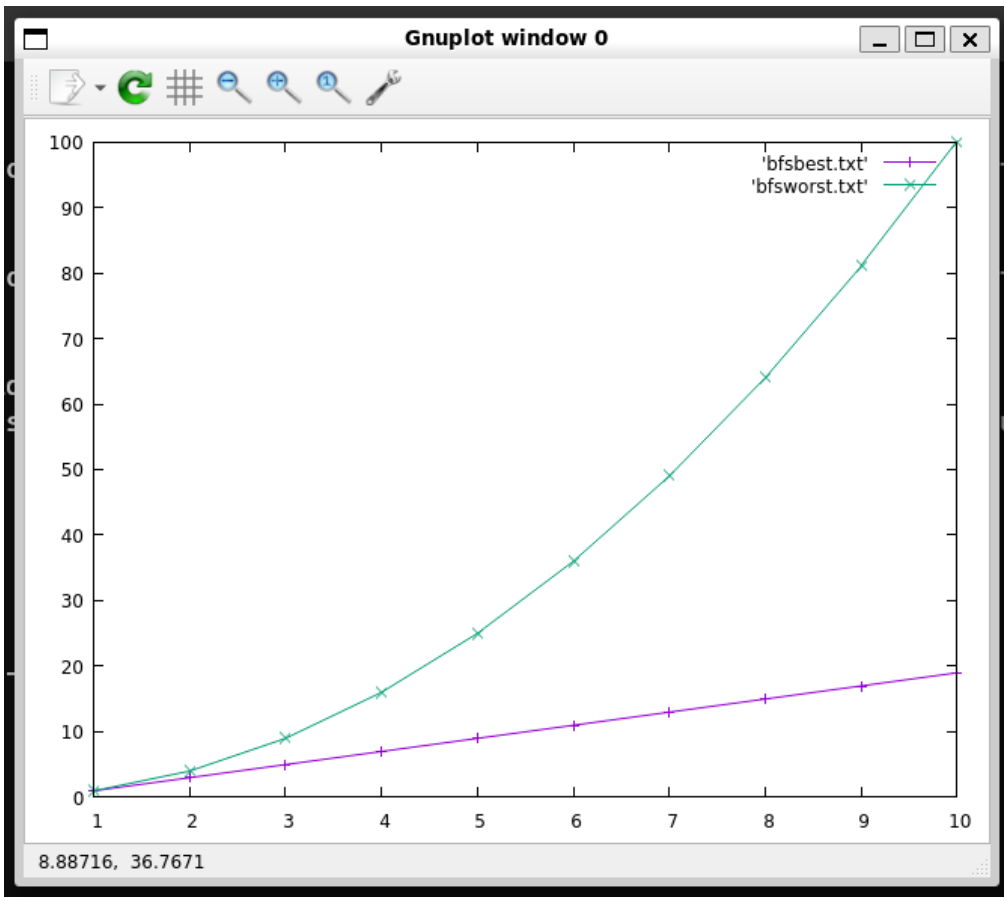
BEST CASE:

1	1	1
2	2	3
3	3	5
4	4	7
5	5	9
6	6	11
7	7	13
8	8	15
9	9	17
10	10	19

WORST CASE:

1	1	1
2	2	4
3	3	9
4	4	16
5	5	25
6	6	36
7	7	49
8	8	64
9	9	81
10	10	100

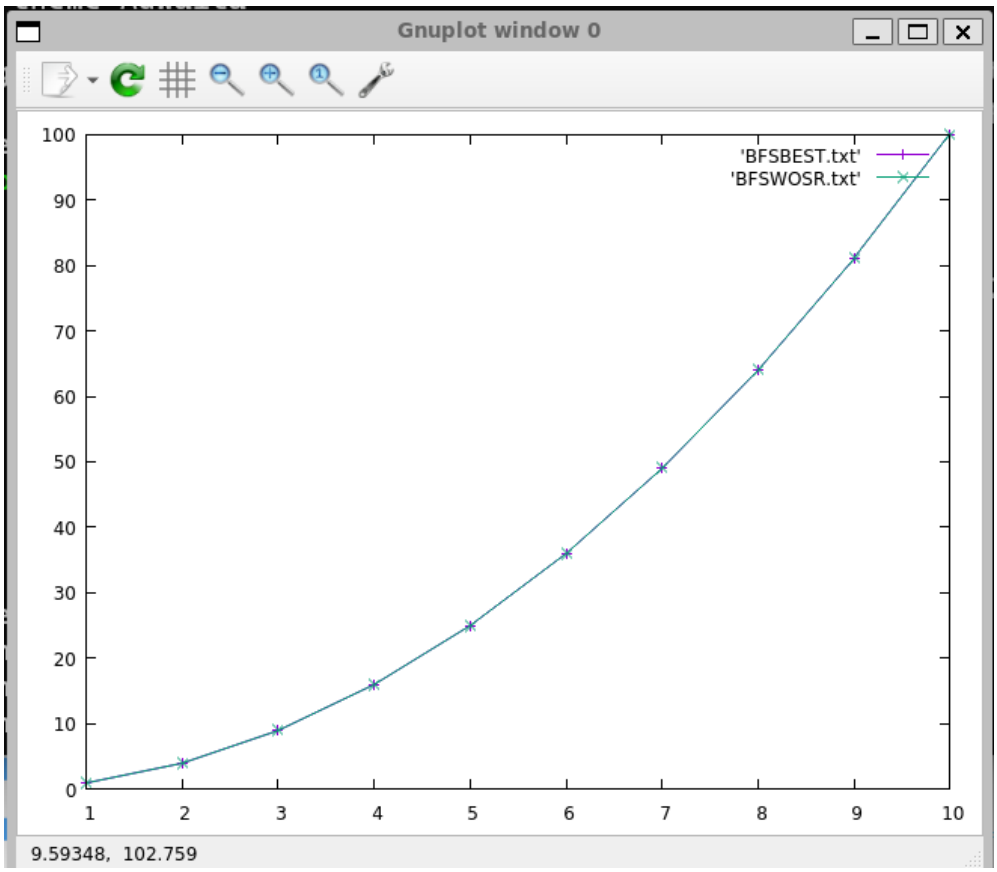
Graph:



WITH ADJACENCY MATRIX:

1	1	1
2	2	4
3	3	9
4	4	16
5	5	25
6	6	36
7	7	49
8	8	64
9	9	81
10	10	100

GRAPH:



10.Implement DFS based algorithm to list the vertices of a directed graph in Topological ordering. Perform its analysis giving minimum 5 graphs with different number of vertices and edges. (starting with 4 vertices). Note: while showing correctness, input should be given for with and without solution.

TESTER:

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 100
int graph[MAX][MAX], visited[MAX],path[MAX], count=0;
int stack[MAX], top=-1;
int c=0;

void dfs(int n, int start) {
   visited[start] = 1;
   path[start] =1;
   for(int i=0; i<n; i++)
     {
        if(graph[start][i] && visited[i]==1&& path[i]==1)
           c=1 ;
      if(graph[start][i] && visited[i]==0)
         dfs(n, i);
      }
     path[start]=0;
   stack[++top] = start;
}

void main()
 {
```

```
    int n;
    printf("\nEnter the number of vertices:\n");
    scanf("%d", &n);
    printf("\nEnter the adjacency matrix:\n");
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++)
            scanf("%d", &graph[i][j]);
        visited[i] = 0;
    }

    printf("\nTopological Order:\n");
    for(int i=0; i<n; i++) {
        if(visited[i] == 0)
            dfs(n, i);
    }
    if(c==1)
    {
      printf("IT HAS A LOOP SO NO TOPOLOGICAL ORDER\n");
      return ;
    }
    for(int i=0; i<n; i++) {
        printf(" --> %c", stack[i]+65);
    }
}
```

PLOTTER:

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 100

int graph[MAX][MAX], visited[MAX],path[MAX], count=0;
int stack[MAX], top=-1;
int c=0;

void dfs(int n, int start) {
   visited[start] = 1;
   path[start] =1;
   for(int i=0; i<n; i++)
     {
        count++;
        if(graph[start][i] && visited[i]==1&& path[i]==1)
           c=1 ;
       if(graph[start][i] && visited[i]==0)
          dfs(n, i);
      }
      path[start]=0;
   stack[++top] = start;
}

void ploter(int k)
{
```

```
  FILE *f1= fopen("BFSBEST.txt", "a");
  FILE *f2=fopen("BFSWOSR.txt", "a");
  int v,start;
  for(int i=1;i<=10;i++)
 {
  v=i;
 int *arr[v];
 for(int i=0;i<v;i++)
 arr[i]=(int *)malloc(sizeof(int)*v);


if(k==0)
 {
 for(int i=0;i<v;i++)
{

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

     if(i!=j)
     {
      arr[i][j] =1;
     }
     else
     arr[i][j] =0;
  }
}
```

```
 }

if(k==1)
{
   for(int i=0;i<v;i++)
   {
      for(int j=0;j<v;j++)
       arr[i][j] =0;
   }
   for(int i=0;i<v-1;i++)
   {
         arr[i][i+1]=1;
   }

}
count=0;
    for(int i=0; i<v; i++) {
      if(visited[i] == 0)
          dfs(v, i);
    }
         if(k==0)
        fprintf(f2,"%d\t%d\n",v,count);
        else
         fprintf(f1,"%d\t%d\n",v,count);
        // printf("%d\t%d\n",v,orderCount);
```

```
  }

  fclose(f1);
  fclose(f2);

}

void main()
{

   for(int i=0; i<2;i++)
   ploter(i);
}
```

Output:

```
THE ADJACENCY MATRIX IS:
0 1 1
1 0 1
1 1 0

Topological Order:
IT HAS A LOOP SO NO TOPOLOGICAL ORDER
hp@LAPTOP-28MPKT32:~$
```

```
0
THE ADJACENCY MATRIX IS:
0 0 1 0 0
0 0 1 0 0
0 0 0 1 1
0 0 0 0 1
0 0 0 0 0

Topological Order:
 --> E --> D --> C --> A --> B
hp@LAPTOP-28MPKT32:~$
```

Linked List:

```
#include<stdio.h>
#include<stdlib.h>

 struct node
{
   int info;
   struct node *next;
};

int dcount=0;
struct Graph{
   int vertices;
   int edges;
    int * visit;
    int *path;
    struct node ** adjLists;
};
  typedef struct node * Node;
  Node createnode(int n)
  {
     Node nn=(Node)malloc(sizeof(struct node));
     nn->info=n;
     nn->next=NULL;
    return nn;
  }
```

```
int stack[200];
int top=-1;
 struct Graph* createGraph(int vertices) {
  struct Graph* graph = malloc(sizeof(struct Graph));
  graph->vertices = vertices;

  graph->adjLists = malloc(vertices * sizeof(struct node*));

  graph->visit = malloc(vertices * sizeof(int));
  graph->path = malloc(vertices * sizeof(int));

  int i;
  for (i = 0; i < vertices; i++) {
    graph->adjLists[i]= NULL;
    graph->visit[i] = 0;
  }
  return graph;
}

int count=0,iscyclic=0;

void DFS(struct Graph* graph, int vertex,int parent) {
  struct node* adjList = graph->adjLists[vertex];
  struct node* temp = adjList;

  count++;
  graph->visit[vertex] = 1;
```

```
  graph->path[vertex] = 1;

  while (temp != NULL)
  {

    int connectedVertex = temp->info;
     dcount++;
    if (graph->visit[connectedVertex]==1&&graph->path[connectedVertex]==1)
        {
            iscyclic=1;
        }
      if(graph->visit[connectedVertex] == 0)
          DFS(graph,connectedVertex,vertex);

    temp = temp->next;
  }
  dcount++;
   graph->path[vertex] =0;
    stack[++top] = vertex;

}

void DFS1(struct Graph* graph, int vertex,int parent) {
  struct node* adjList = graph->adjLists[vertex];
  struct node* temp = adjList;

  count++;
```

```
  graph->visit[vertex] = 1;
  graph->path[vertex] = 1;

  while (temp != NULL)
  {
   int connectedVertex = temp->info;
   if (graph->visit[connectedVertex]==1&&graph->path[connectedVertex]==1)
       {
           iscyclic=1;
            return;
       }
     if(graph->visit[connectedVertex] == 0)
         DFS(graph,connectedVertex,vertex);

   temp = temp->next;
  }
   graph->path[vertex] =0;
   stack[++top] = vertex;

}



void tester()
{
   int n;
   printf("ENTER THE NUUMBER OF VERTICES\n");
```

```
   scanf("%d",&n);
   struct Graph*   g=createGraph(n);
   Node temp;
   int key;
    top=-1;

  printf("Enter the adjacency LIST \n");
  for(int i=0;i<g->vertices;i++)
{
  printf("Enter 1 for the vertices adjacent to vertex %c\n",i+65);
   for(int j=0;j<g->vertices;j++)
  {
      printf("\nVertex %c : ",g->vertices-j-1+65);
      scanf("%d",&key);
      if(key==1)
      {
          Node nn=createnode(g->vertices-j-1);
          nn->next = g->adjLists[i];
          g->adjLists[i] = nn;
      }
  }
}


   for(int i=0;i<g->vertices;i++)
  {
    temp=g->adjLists[i];
```

```
  printf("THE VERTEX ADJACENT TO %c : ",i+65);
   while(temp!=NULL)
    {
     printf("%c ",temp->info+65);
     temp=temp->next;
    }
  printf("\n");
 }
int dfscount=0;
count=0;
iscyclic=0;
 printf("\nDFS TRAVERSAL STARTING FROM NODE %C\n",65);
 DFS1(g,0,-1);
 dfscount++;
 int start=1;
    while(count!=g->vertices)
      {
        if(g->visit[start]!=1)
         {
           printf("\n");
           DFS1(g,start,-1);
           dfscount++;
         }
        start++;
      }
 if(iscyclic==1)
 {
```

```
      printf("\nTHE GRAPH  HAS A  CYCLE \n");
    }
    else
    {
    printf("\nTHE TOPOLOGICAL SORT IS:\n");
    for(int i=0;i<g->vertices;i++)
     printf("-->%c ", stack[i]+65);
    }
    free(g);
  }

void ploter(int k)
{

   FILE *f1,*f2;
   f1=fopen("TOPODFSWROST.txt","a");
   f2=fopen("TOPODFSBEST.txt","a");
    for(int i=1;i<=20;i++)
    {
      int n=i;

     struct Graph*   g=createGraph(n);
     Node temp;
     int key;


 if(k==0)
```

```
  for(int i=0;i<g->vertices;i++)
{

   for(int j=0;j<g->vertices;j++)
   {

      if(i!=g->vertices-1-j)
      {
          Node nn=createnode(g->vertices-j-1);
          nn->next = g->adjLists[i];
          g->adjLists[i] = nn;
      }
   }
}

if(k==1)
{
    for(int i=0;i<g->vertices-1;i++)
    {
          Node nn=createnode(i+1);
          nn->next = g->adjLists[i];
          g->adjLists[i] = nn;
    }

}
  count=0;
  dcount=0;
```

```
int dfscount=0;
DFS(g,0,-1);
dfscount++;
int start=1;
   while(count!=g->vertices)
    {
      if(g->visit[start]!=1)
       {
         printf("\n");
         DFS(g,start,-1);
         dfscount++;
       }
      start++;
    }

    if(k==0)
    fprintf(f2,"%d\t%d\n",n,count);
    else
     fprintf(f2,"%d\t%d\n",n,count);


     free(g);
 }


fclose(f1);
fclose(f2);
```

```
}



void main()
{
   for(;;)
   {
      int key;
      printf("ENTER THE CHOICE 1.TO TEST \n2.TO PLOT\nOTHER TO
EXIT\n");
      scanf("%d",&key);


       switch(key)
       {
         case 1:tester();break;
         case 2:for(int i=0;i<2;i++)
                ploter(i);
                break;
         default:exit(1);
       }


   }
}
```

LINKED LIST:

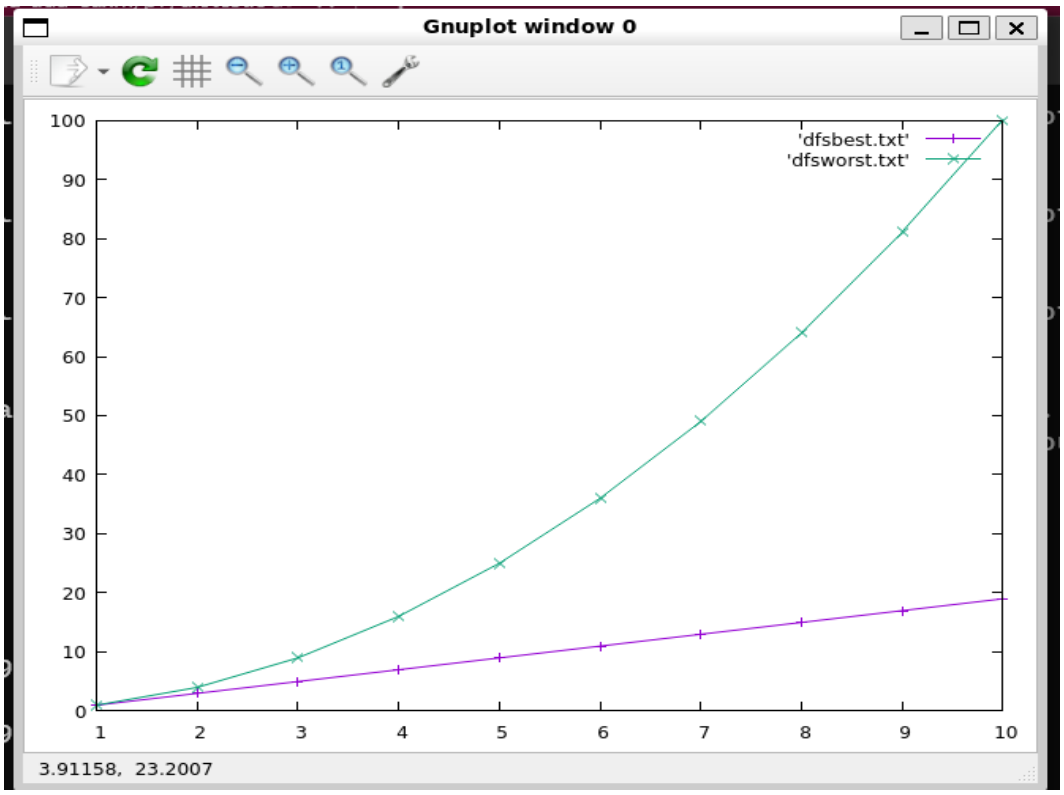
Best case:

```
 1 1         1
 2 2         3
 3 3         5
 4 4         7
 5 5         9
 6 6         11
 7 7         13
 8 8         15
 9 9         17
10 10        19
```

Worst case:

```
 1 1         1
 2 2         4
 3 3         9
 4 4         16
 5 5         25
 6 6         36
 7 7         49
 8 8         64
 9 9         81
10 10        100
11
```

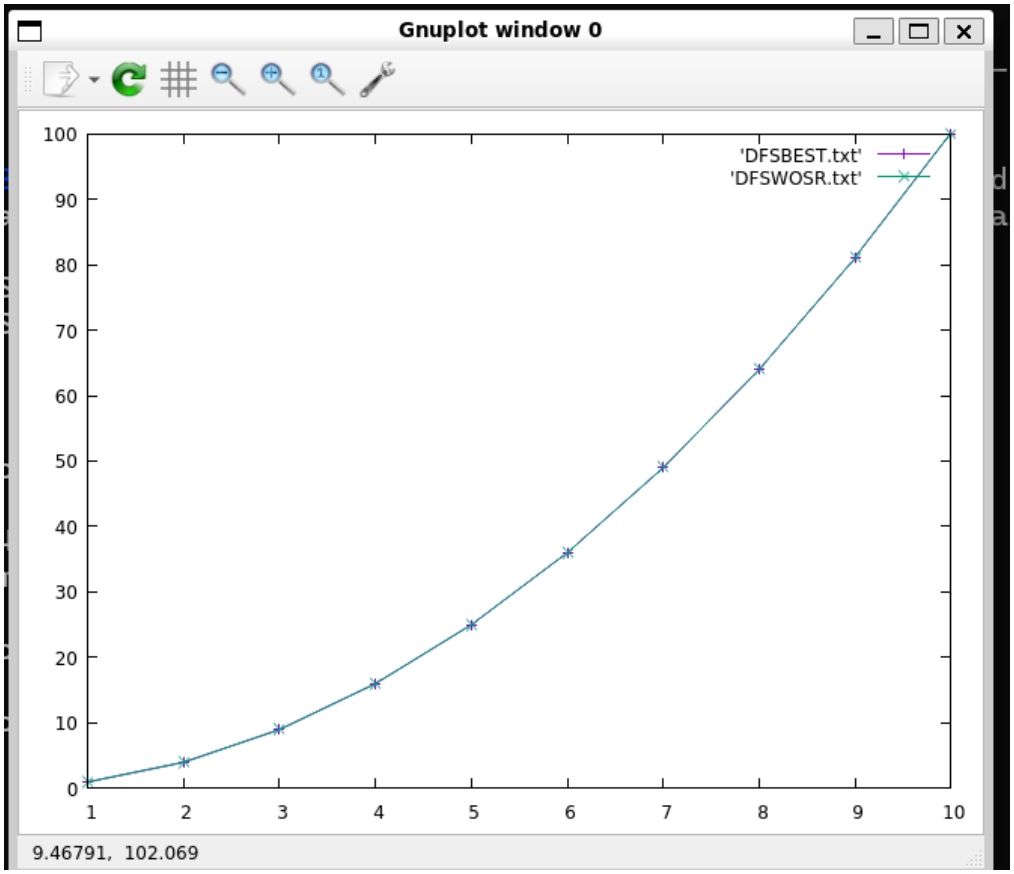
Graph:



WITH ADJACENCY MATRIX:

```
 1 1         1
 2 2         4
 3 3         9
 4 4         16
 5 5         25
 6 6         36
 7 7         49
 8 8         64
 9 9         81
10 10        100
11
```

Graph:



9. .Implement source removal algorithm to list the vertices of a directed graph in Topological ordering. Perform its analysis giving minimum 5 graphs with different number of vertices and edges. (starting with 4 vertices). Note: Use efficient method to identify the source vertex. While showing correctness, Input should be given for with and without solution.

MATRIX:

```
#include<stdio.h>
#include<stdlib.h>
  int count=0;


typedef struct queue
{
int f,r,*arr,cnt;
}QUE;


int s[10];


 void indegree(int *a[],int v,int inq[],QUE *temp,int flag[])
{
      for(int i=0;i<v;i++)
     {
      for(int j=0;j<v;j++)
       {
         if(a[j][i]==1)
        inq[i]=inq[i]+1;
       }
          if(inq[i]==0)
          {
            temp->r=(temp->r+1)%v;
```

```
            temp->arr[temp->r]=i;
            temp->cnt=temp->cnt+1;
             flag[i]=1;
           }
       }
}

void sourceremove(int *a[],int v,QUE *temp,int inq[],int flag[])
{
int cnt=0;
while(temp->cnt!=0)
{
  int source=temp->arr[temp->f];
   temp->f=(temp->f+1)%v;
   s[cnt]=source;
   temp->cnt=temp->cnt-1;
    cnt++;
   for(int i=0;i<v;i++)
   {
    if(a[source][i]==1)
      inq[i]=inq[i]-1;
      if(inq[i]==0&&flag[i]==0)
     {
       temp->r=(temp->r+1)%v;
       temp->arr[temp->r]=i;
       temp->cnt=temp->cnt+1;
        flag[i]=1;
```

```
    }
  }
}

if(cnt!=v)
{
  printf("Cycles exist no topological sorting possible\n");
}
else
{

  printf("The topological sorting is\n");
  for(int i=0;i<v;i++)
   printf("%c\t",s[i]+65);

}
}

void main()
{
  int v;
  printf("Enter number of vertices\n"); scanf("%d",&v);
  int *arr[v];
  for(int i=0;i<v;i++)
  arr[i]=(int *)malloc(sizeof(int)*v);
printf("Enter the adjacency matrix\n");
```

```
  for(int i=0;i<v;i++)
{
  //printf("Enter 1 for the vertices adjacent to vertex %c\n",i+65);
   for(int j=0;j<v;j++)
  {
    //printf("\nVertex %c : ",j+65);
     scanf("%d",&arr[i][j]);
  }
}
printf("\n");

     printf("Adjacency matrix\n");
   for(int i=0;i<v;i++)
  {
    for(int j=0;j<v;j++)
    {
     printf("%d\t",arr[i][j]);
    }
     printf("\n");
   }
printf("\n");
  QUE q;
  q.f=0;
  q.r=-1;
  q.cnt=0;
   q.arr=(int*)malloc(sizeof(int)*v);
```

```
 int *inq=(int *)malloc(sizeof(int)*v);
  for(int i=0;i<v;i++)
  inq[i]=0;
 int *flag=(int *)malloc(sizeof(int)*v);
 for(int i=0;i<v;i++)
 flag[i]=0;

  indegree(arr,v,inq,&q,flag);
  sourceremove(arr,v,&q,inq,flag);

printf("\n");
}
```

LINKED LIST:

```
#include<stdio.h>
 #include<stdlib.h>

int count1,count2,count3;
  struct node
{
  int info;
  struct node *next;
};
struct Graph{
  int vertices;
  int edges;
   int * visit;
   int *path;
```

```
  struct node ** adjLists;
};
 typedef struct node * Node;

 Node createnode(int n)
 {
   Node nn=(Node)malloc(sizeof(struct node));
   nn->info=n;
   nn->next=NULL;
  return nn;
 }
 int count=0;


 int stack[100];
 int top=-1;
 struct Graph* createGraph(int vertices) {
 struct Graph* graph = malloc(sizeof(struct Graph));
 graph->vertices = vertices;
 graph->edges=0;
 graph->adjLists = malloc(vertices * sizeof(struct node*));

 graph->visit = malloc(vertices * sizeof(int));
 graph->path = malloc(vertices * sizeof(int));

 int i;
 for (i = 0; i < vertices; i++) {
```

```
    graph->adjLists[i]= NULL;
    graph->visit[i] = 0;
  }
  return graph;
}



typedef struct queue
{
int f,r,*arr,cnt;
}QUE;
int s[100];

  void indegree(  struct Graph* graph ,int inq[],QUE *temp,int flag[])
{
   Node adj;
      for(int i=0;i<graph->vertices;i++)
    {
       adj=graph->adjLists[i];
       while(adj!=NULL)
       {
           int k=adj->info;
           inq[adj->info]++;
          adj=adj->next;
           count1++;
       }
       count1++;
```

```
    }
    for(int i=0;i<graph->vertices;i++)
    {
        if(inq[i]==0)
        {
         temp->r=(temp->r+1)%graph->vertices;
          temp->arr[temp->r]=i;
          temp->cnt=temp->cnt+1;
           flag[i]=1;
          // count2++;
         }
          count2++;
    }
}

void sourceremove( struct Graph* graph,QUE *temp,int inq[],int flag[])
{
int cnt=0;
while(temp->cnt!=0)
{
  int source=temp->arr[temp->f];
   temp->f=(temp->f+1)%graph->vertices;
   s[cnt]=source;
   temp->cnt=temp->cnt-1;
    cnt++;
    Node adj;
    adj=graph->adjLists[source];
```

```
        while(adj!=NULL)
        {
            int k=adj->info;
            inq[k]--;
            adj=adj->next;

           count3++;
            if(inq[k]==0&&flag[k]==0)
           {
            temp->r=(temp->r+1)%graph->vertices;
             temp->arr[temp->r]=k;
             temp->cnt=temp->cnt+1;
              flag[k]=1;
             // count2++;
           }
        }
          count3++;
}

}


int max(int num1,int num2,int num3)
{
    if(num1>num2&&num1>num3)
     return num1;
   if(num2>num1&&num2>num3)
```

```
        return num2;
        return num3;
}

void ploter(int k)
{

   FILE *fdata,*f1,*f2;
   fdata = fopen("GRAPHDATA.txt","a");
   f1=fopen("TSRCWROST.txt","a");
   f2=fopen("TSRCBEST.txt","a");
    for(int i=1;i<=100;i++)
    {
      int n=i;

     struct Graph*   g=createGraph(n);
     Node temp;
     int key;


 if(k==0)
  for(int i=0;i<g->vertices;i++)
{

   for(int j=0;j<g->vertices;j++)
   {
```

```
        if(i!=g->vertices-1-j)
        {
            Node nn=createnode(g->vertices-j-1);
            nn->next = g->adjLists[i];
            g->adjLists[i] = nn;


        }
    }
}

if(k==1)
{
    for(int i=0;i<g->vertices-1;i++)
    {
            Node nn=createnode(i+1);
            nn->next = g->adjLists[i];
            g->adjLists[i] = nn;
    }

}
   count1=0;
   count2=0;
   count3=0;
  QUE q;
  q.f=0;
  q.r=-1;
```

```
 q.cnt=0;
  q.arr=(int*)malloc(sizeof(int)*g->vertices);

 int *inq=(int *)malloc(sizeof(int)*g->vertices);
  for(int i=0;i<g->vertices;i++)
  inq[i]=0;
int *flag=(int *)malloc(sizeof(int)*g->vertices);
for(int i=0;i<g->vertices;i++)
flag[i]=0;

  indegree(g,inq,&q,flag);
  sourceremove(g,&q,inq,flag);
 int max1= max(count1,count2,count3);
  fprintf(fdata,"%d \t %d\t %d\t%d\n",n+g->edges,count1,count2,count3);
  if(k==0)
   fprintf(f1,"%d \t %d\n",n+g->edges,max1);
   else
   fprintf(f2,"%d \t %d\n",n+g->edges,max1);
  free(g);

}
fclose(fdata);
fclose(f1);
fclose(f2);
}
```

```
void sourceremove1( struct Graph* graph,QUE *temp,int inq[],int flag[])
{
int cnt=0;
while(temp->cnt!=0)
{
 int source=temp->arr[temp->f];
  temp->f=(temp->f+1)%graph->vertices;
  s[cnt]=source;
  temp->cnt=temp->cnt-1;
   cnt++;
   Node adj;
   adj=graph->adjLists[source];
      while(adj!=NULL)
      {
          int k=adj->info;
          inq[k]--;
          adj=adj->next;



          if(inq[k]==0&&flag[k]==0)
          {
          temp->r=(temp->r+1)%graph->vertices;
           temp->arr[temp->r]=k;
           temp->cnt=temp->cnt+1;
            flag[k]=1;
          }
      }
```

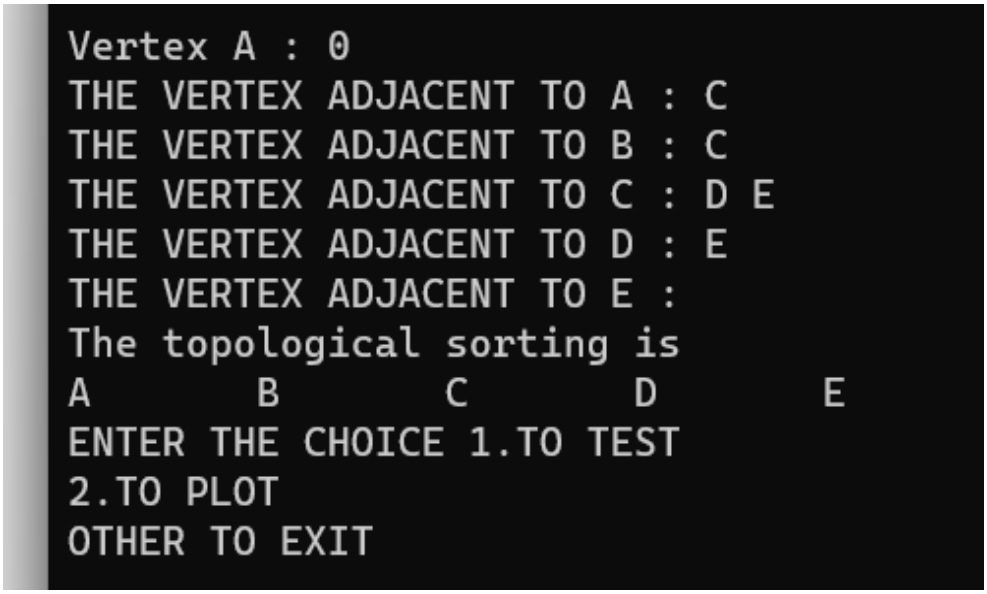
```
}

if(cnt!=graph->vertices)
{
  printf("Cycles exist no topological sorting possible\n");
}
else
{

  printf("The topological sorting is\n");
  for(int i=0;i<graph->vertices;i++)
   printf("%c\t",s[i]+65);

}
}

void tester()
{
     int n;
     printf("ENTER THE NUUMBER OF VERTICES\n");
    scanf("%d",&n);
   struct Graph*   g=createGraph(n);
   Node temp;
  int key;


printf("Enter the adjacency LIST \n");
```

```
 for(int i=0;i<g->vertices;i++)
{
 printf("Enter 1 for the vertices adjacent to vertex %c\n",i+65);
  for(int j=0;j<g->vertices;j++)
 {
     printf("\nVertex %c : ",g->vertices-j-1+65);
     scanf("%d",&key);
     if(key!=0)
     {
         Node nn=createnode(g->vertices-j-1);
         nn->next = g->adjLists[i];
         g->adjLists[i] = nn;

     }
 }
}


  for(int i=0;i<g->vertices;i++)
 {
   temp=g->adjLists[i];
   printf("THE VERTEX ADJACENT TO %c : ",i+65);
    while(temp!=NULL)
     {
      printf("%c ",temp->info+65);
      temp=temp->next;
     }
```

```
    printf("\n");
   }
  QUE q;
  q.f=0;
  q.r=-1;
  q.cnt=0;
   q.arr=(int*)malloc(sizeof(int)*g->vertices);

  int *inq=(int *)malloc(sizeof(int)*g->vertices);
   for(int i=0;i<g->vertices;i++)
   inq[i]=0;
 int *flag=(int *)malloc(sizeof(int)*g->vertices);
 for(int i=0;i<g->vertices;i++)
 flag[i]=0;

   indegree(g,inq,&q,flag);
   sourceremove1(g,&q,inq,flag);

printf("\n");
free(g);
}


void main()
{
    for(;;)
    {
```

```
    int key;
    printf("ENTER THE CHOICE 1.TO TEST \n2.TO PLOT\nOTHER TO EXIT\n");
    scanf("%d",&key);

     switch(key)
     {
      case 1:tester();break;
      case 2:for(int i=0;i<2;i++)
            ploter(i);
            break;
      default:exit(1);
     }

  }
}
```

Output



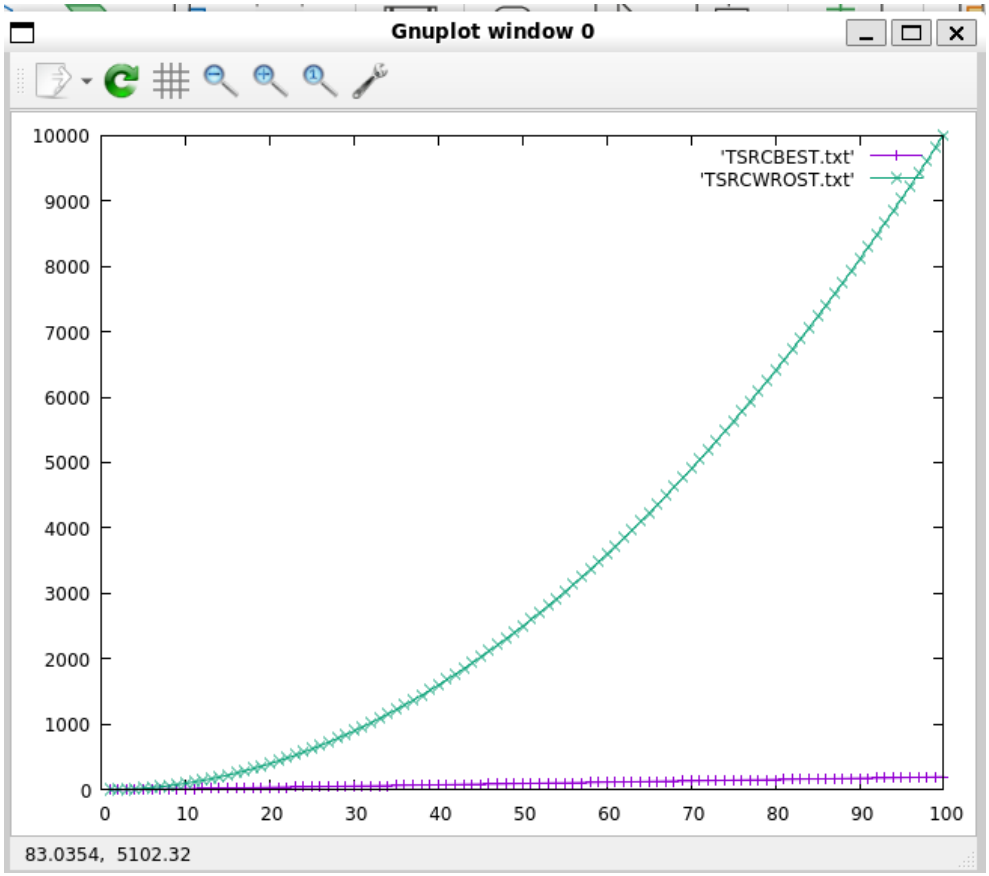
```
Vertex A : 1
THE VERTEX ADJACENT TO A : B C
THE VERTEX ADJACENT TO B : A C
THE VERTEX ADJACENT TO C : A B
Cycles exist no topological sorting possible

ENTER THE CHOICE 1.TO TEST
2.TO PLOT
OTHER TO EXIT
```

Best case:

1	1	1
2	2	3
3	3	5
4	4	7
5	5	9
6	6	11
7	7	13
8	8	15
9	9	17
10	10	19
11	11	21
12	12	23
13	13	25
14	14	27
15	15	29
16	16	31
17	17	33
18	18	35
19	19	37
20	20	39

Worst case:

1	1	1
2	2	4
3	3	9
4	4	16
5	5	25
6	6	36
7	7	49
8	8	64
9	9	81
10	10	100
11	11	121
12	12	144
13	13	169
14	14	196
15	15	225
16	16	256
17	17	289
18	18	324
19	19	361
20	20	400



11.Implement heap sort algorithm with bottom-up heap construction. Perform its analysis by generating best case and worst case data.

TESTER:

```
/*heap sort with recursion */
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

int count,count2=0;
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
```

```
    *b = temp;
    return;
}

void heapify(int *heap, int n, int root) {
    int largest = root;
    int left = 2*root+1;
    int right = 2*root+2;
    if(left < n )
    {
         count++;
    if(heap[left] > heap[largest]) {
       largest = left;
    }
    }
    if(right < n )
    {
         count++;
    if(heap[right] > heap[largest]) {
       largest = right;
    }
    }
    if(largest != root) {
       swap(&heap[root], &heap[largest]);
       heapify(heap, n, largest);
    }
}
```

```
void heapSort(int *heap, int n) {
    for(int i = (n/2)-1; i>=0; i--) {
        heapify(heap, n, i);
    }
     count2=count;
     count=0;
    for(int i = n-1; i>=0; i--) {
        swap(&heap[0], &heap[i]);
        heapify(heap, i, 0);
    }

}

int max(int a, int b) {

    int temp =a>b ? a:b;
    return temp;
}
void main()
{
  int *arr, n;
   printf("ENTER THE NUMBER OF ELEMENTS\n");
   scanf("%d",&n);

   arr=(int *)malloc(sizeof(int)*n);
```

```
   printf("ENTER THE ELEMENTS OF THE ARRAY\n");
      for(int i=0;i<n;i++)
       scanf("%d",&arr[i]);

    printf("THE ELEMENTS OF THE ARRAY BEFORE SORTING\n");
     for(int i=0;i<n;i++)
        printf("%d ",arr[i]);
     printf("\n");

      heapSort(arr,n);

      printf("THE ELEMENTS OF THE ARRAY BEFORE SORTING\n");
     for(int i=0;i<n;i++)
       printf("%d ",arr[i]);
       printf("\n");
       printf("\n");
}
```

PLOTTER:

```
/*heap sort with recursion */
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

int count,count2=0;
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
```

```
    *b = temp;
    return;
}

void heapify(int *heap, int n, int root) {
    int largest = root;
    int left = 2*root+1;
    int right = 2*root+2;
    if(left < n )
    {
         count++;
    if(heap[left] > heap[largest]) {
       largest = left;
    }
    }
    if(right < n )
    {
         count++;
    if(heap[right] > heap[largest]) {
       largest = right;
    }
    }
    if(largest != root) {
       swap(&heap[root], &heap[largest]);
       heapify(heap, n, largest);
    }
}
```

```
void heapSort(int *heap, int n) {
    for(int i = (n/2)-1; i>=0; i--) {
        heapify(heap, n, i);
    }
     count2=count;
     count=0;
    for(int i = n-1; i>=0; i--) {
        swap(&heap[0], &heap[i]);
        heapify(heap, i, 0);
    }

}

int max(int a, int b) {

    int temp =a>b ? a:b;
    return temp;
}

void plotter()
{

   int *arr,n;
   srand(time(NULL));
   FILE *f1,*f2,*f3;
```

```
f1=fopen("HEAPSORTBEST.txt","a");
f2=fopen("HEAPSORTWORST.txt","a");
f3=fopen("HEAPSORTAVG.txt","a");
 n=100;

 while(n<=1000)
 {
   arr=(int *)malloc(sizeof(int)*(n+1));
   for(int i=0;i<n;i++)
   *(arr+i)=n-i+1;
   count=0;
    //best case
     heapSort(arr,n);
     count=max(count,count2);
    fprintf(f1,"%d\t%d\n",n,count);
    //printf("%d\t%d\n",n,count);

 //worst case
  count=0;
  for(int i=0;i<n;i++)
  *(arr+i)=i+1;
    heapSort(arr,n);
        count=max(count,count2);
  fprintf(f2,"%d\t%d\n",n,count);
  //printf("%d\t%d\n",n,count);
```

```
    //AVG case
     for(int i=0;i<n;i++)
     *(arr+i)=rand()%n;
     count=0;
     heapSort(arr,n);
        count=max(count,count2);
     fprintf(f3,"%d\t%d\n",n,count);
     // printf("%d\t%d\n",n,count);

     n=n+100;
     free(arr);
   }
   fclose(f1);
   fclose(f2);
   fclose(f3);
}


void main()
{

        plotter();
        printf("THE DATA ENETERED IN TO THE FILE\n");
}
```

OUTPUT:

```
2
THE ELEMENTS OF THE ARRAY BEFORE SORTING
-9 6 7 -100 2
THE ELEMENTS OF THE ARRAY BEFORE SORTING
-100 -9 2 6 7
```

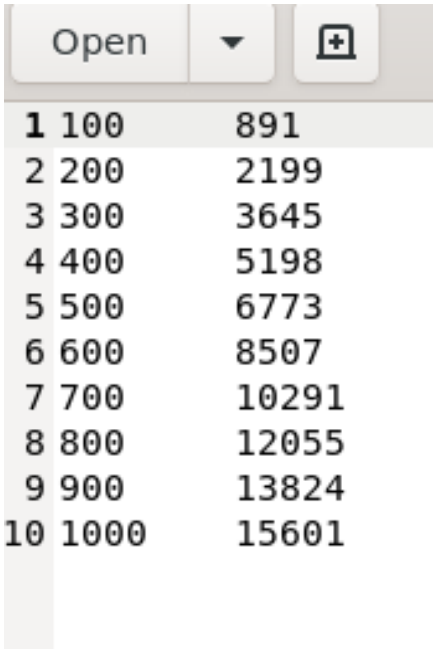
```
5
THE ELEMENTS OF THE ARRAY BEFORE SORTING
-9 -8 -7 -6 -5
THE ELEMENTS OF THE ARRAY BEFORE SORTING
-9 -8 -7 -6 -5
```

```
-9
THE ELEMENTS OF THE ARRAY BEFORE SORTING
-5 -6 -7 -8 -9
THE ELEMENTS OF THE ARRAY BEFORE SORTING
-9 -8 -7 -6 -5
```

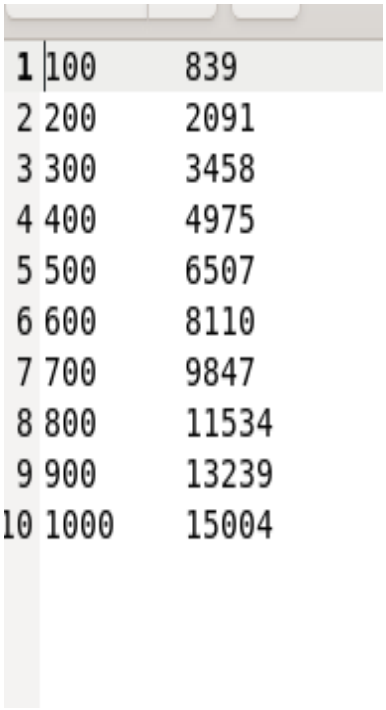
BEST CASE:

```
1  100     845
2  200     2076
3  300     3460
4  400     4953
5  500     6511
6  600     8119
7  700     9813
8  800     11494
9  900     13283
10 1000    14966
```

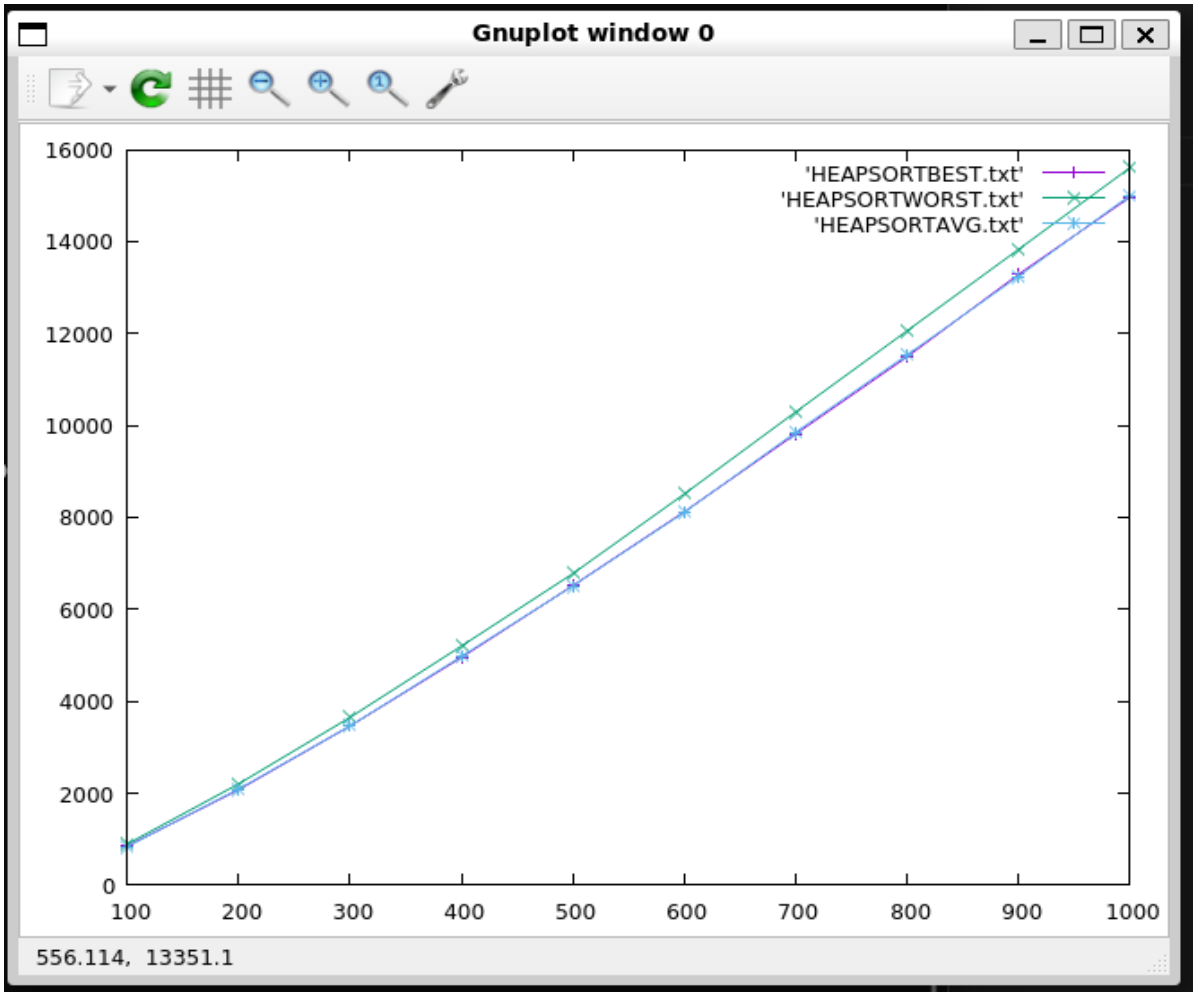
WORST CASE:



AVG CASE:



GRAPH:



10. a) Implement Warshall's algorithm to find the transitive closure of a directed graph and perform its analysis for different inputs.

```
#include <stdio.h>

int graph[40][40], n, count=0;

void createGraph(){

        printf("No. of vertices>> ");
```

```
        scanf("%d", &n);
        printf("Enter adjacency matrix:\n");
        for(int i=0;i<n;i++){
                for(int j=0;j<n;j++){
                        scanf("%d",&graph[i][j]);
                }
        }
}
void main(){
        createGraph();
        for(int k=0;k<n;k++){
                for(int i=0;i<n;i++){
                        if(graph[i][k]==1){
                                for(int j=0;j<n;j++){
                                        count++;
                                        if(graph[k][j]==1){
                                                graph[i][j] = 1;
                                        }
                                }
                        }
```

```
            }
        }
printf("Applying Warshall's Algorithm\n");
printf("Transitive Closure Matrix:\n");
for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            printf("%d ",graph[i][j]);
            }
            printf("\n");
        }
printf("Operation Count: %d\n",count);
}
```

```
No. of vertices>> 4
Enter adjacency matrix:
0 1 0 0
0 0 0 1
0 0 0 0
1 0 1 0
Applying Warshall's Algorithm
Transitive Closure Matrix:
1 1 1 1
1 1 1 1
0 0 0 0
1 1 1 1
Operation Count: 28
```

```
No. of vertices>> 4
Enter adjacency matrix:
0 1 1 1
1 0 1 1
1 1 0 1
1 1 1 0
Applying Warshall's Algorithm
Transitive Closure Matrix:
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
Operation Count: 60
```

Plotter

```
/* program to implement the Warshall's Algorithm*/

#include<stdio.h>

#include<stdlib.h>

int graph[100][100];

int counter=0;

void warshall (int n)

{

   for(int k=1; k<=n; k++)

   {

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

      {

        if(graph[i][k]!=0)

         {
```

```
                for(int j=1; j<=n; j++)
                {// graph[i][j] = (graph[i][k] && graph[k][j]));
                 graph[i][j] = (graph[i][j] || (graph[i][k] && graph[k][j]));
                  counter++;
                }
            }
        }
    }
}
void ploter(int c)
{
        FILE *f1=fopen("warshalbest.txt","a");
        FILE *f2=fopen("warshallworst.txt","a");
        for(int i=1;i<=10;i++)
        {
                int n=i;
                if(c==1)
                {
                        for(int i=1;i<=n;i++)
                        {
```

```
                for(int j=1;j<=n;j++)
                {
                if(i!=j)
                {
                graph[i][j] =1;
                }
                else
                graph[i][j] =0;
                }
        }
}
if(c==0)
{
        for(int i=1;i<=n;i++)
        {
                for(int j=1;j<=n;j++)
                        graph[i][j] =0;
        }
        for(int i=1;i<n;i++)
        {
```

```
                graph[i][i+1]=1;
            }
            graph[n][1]=1;
        }
    counter=0;
    warshall(n);
    if(c==0)
        fprintf(f1,"%d\t%d\n",n,counter);
    else
        fprintf(f2,"%d\t%d\n",n,counter);
    }
    fclose(f1);
    fclose(f2);
}
void main()
{
     for(int i=0;i<2;i++)
        ploter(i);
    printf("the graph is plotted\n");
}
```

warshalbest

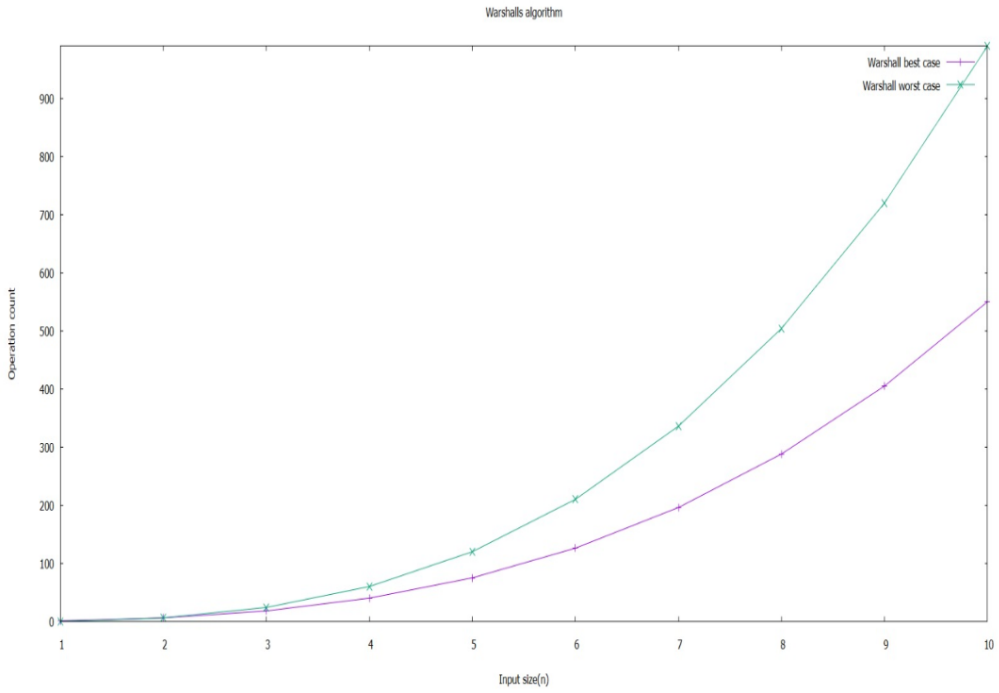
File Edit View

1	1
2	6
3	18
4	40
5	75
6	126
7	196
8	288
9	405
10	550

warshallworst

File Edit View

1	0
2	6
3	24
4	60
5	120
6	210
7	336
8	504
9	720
10	990



b) Implement Floyd's algorithm to find all pair shortest paths for a graph and perform its analysis for different inputs

#include<stdio.h>

int graph[40][40],n,count=0;

```
void creategraph(){
        printf("Number of vertices>>");
         scanf("%d",&n);
        printf("Enter adjacency matrix:\n");
        for(int i=0;i<n;i++){
                for(int j=0;j<n;j++){
                        scanf("%d",&graph[i][j]);
                }
        }
}
void main(){
        creategraph();
        int temp;
        for(int k=0;k<n;k++){
                for(int i=0;i<n;i++){
                        temp=graph[i][k];
                        for(int j=0;j<n;j++){
                                if(graph[i][j]>temp){
                                count++;
                                if(temp+graph[k][j]<graph[i][j])
```

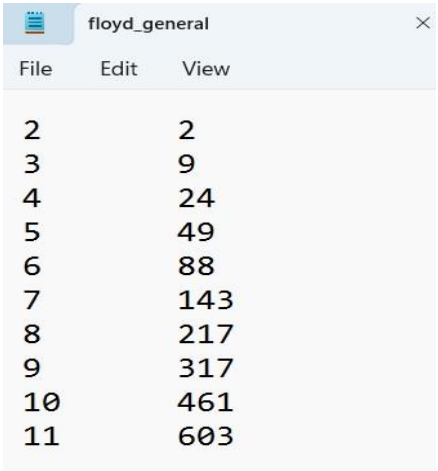
```
                                    graph[i][j]=temp+graph[k][j];
                                }
                            }
                        }
                    }
printf("Applying Floyd's algorithm\n");
printf("all pair shortest path matrix:\n");
for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
                printf("%d  ",graph[i][j]);
                }
        printf("\n");
        }
printf("operation count:%d\n",count);
}
```

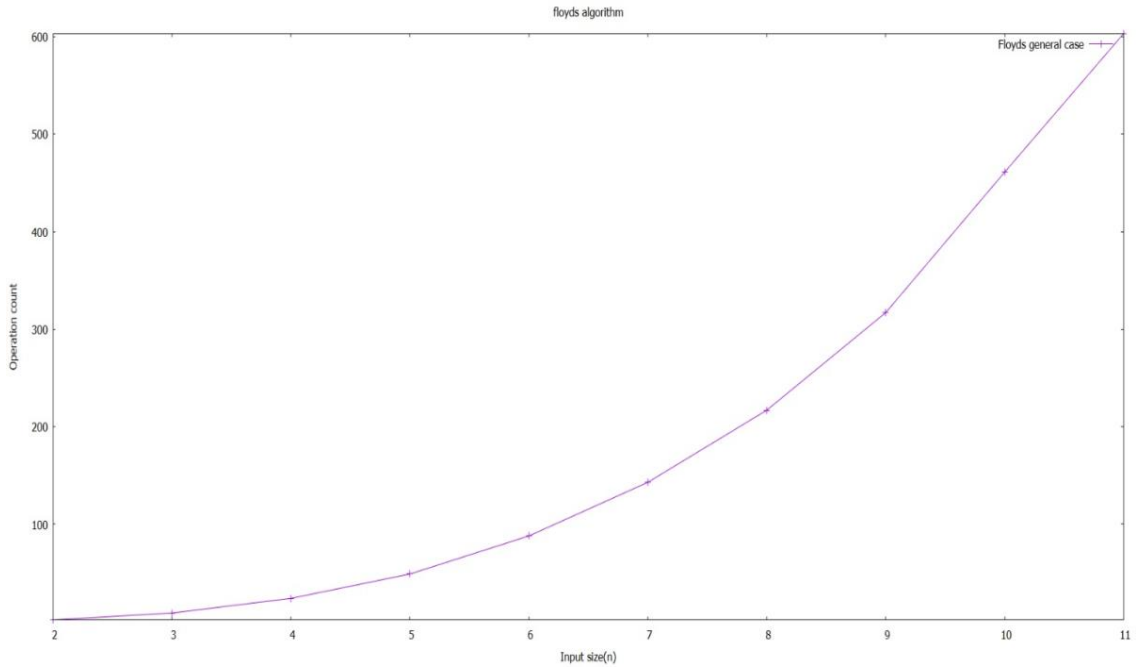
```
Number of vertices>>4
Enter adjacency matrix:
0 99 3 99
2 0 99 99
99 7 0 1
6 99 99 0
Applying Floyd's algorithm
all pair shortest path matrix:
0  10  3  4
2  0  5  6
7  7  0  1
6  16  9  0
operation count:24
```

Plotter

```
#include<stdio.h>
#include<stdlib.h>
int graph[40][40],n,count=0;
void creategraph(int n){
        for(int i=0;i<n;i++){
                for(int j=0;j<n;j++){
                if(i==j) graph[i][j]=0;
                else graph[i][j]=rand();
                }
        }
}
void main(){
   FILE *fp;
   fp=fopen("floyd_general","w");
   for(n=2;n<12;n++)
   {
         count=0;
        creategraph(n);
        int temp;
        for(int k=0;k<n;k++){
                for(int i=0;i<n;i++){
                        temp=graph[i][k];
                                for(int j=0;j<n;j++){
                        if(graph[i][j]>temp){
                                                count++;
                                                if(temp+graph[k][j]<graph[i][j])
```

```
                                    graph[i][j]=temp+graph[k][j];
                                }
                            }
                        }
                    }
        printf("Applying Floyd's algorithm\n");
        printf("all pair shortest path matrix:\n");
        for(int i=0;i<n;i++){
                for(int j=0;j<n;j++){
                printf("%d  ",graph[i][j]);
                }
                printf("\n");
        }
        printf("operation count:%d\n",count);
        fprintf(fp,"%d\t%d\n",n,count);
    }
    fclose(fp);
}
```





11. a) Implement an algorithm to solve Knapsack problem with dynamic programming approach and perform its analysis for different inputs.

```
#include<stdio.h>

#include<stdlib.h>

int t[100][100],v[100],w[100],n,m,i,j;

int max(int a,int b){

        return (a>b)?a:b;

}

int knap(int n,int m)

{

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

        {
```

```
            for(j=0;j<m+1;j++)
            {
                if(i==0||j==0)
                    t[i][j]=0;
                else if(j<w[i])
                    t[i][j]=t[i-1][j];
                else
                    t[i][j]=max(t[i-1][j],v[i]+t[i-1][j-w[i]]);
            }
        }
    return t[n][m];
}
void main()
{
        printf("Number of items:  ");
        scanf("%d",&n);
        printf("Sack capacity:  ");
        scanf("%d",&m);
        printf("Weight\tValue\n");
        for(i=1;i<n+1;i++)
```

```
{
        scanf("%d\t%d",&w[i],&v[i]);
}
printf("Max value %d\n",knap(n,m));
for(int i=0;i<n+1;i++)
{
        for(int j=0;j<m+1;j++)
                printf("%d  ",t[i][j]);
        printf("\n");
}
printf("Composition:\n");
for(int i=n;i>0;i--)
{
        if(t[i][m]!=t[i-1][m])
        {
                printf("%d\t",i);
                m=m-w[i];
        }
}
printf("\n");
```

}

```
Number of items:5
Sack capacity:11
Weight  Value
7       65
4       30
2       5
1       3
5       45
Max value 95
0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  65  65  65  65  65
0  0  0  0  30  30  30  65  65  65  65  95
0  0  5  5  30  30  35  65  65  70  70  95
0  3  5  8  30  33  35  65  68  70  73  95
0  3  5  8  30  45  48  65  68  75  78  95
Composition:
2       1
```

PLOTTER:

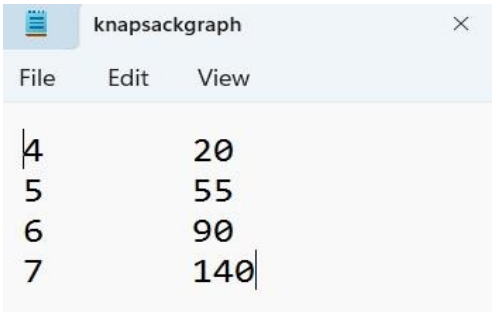
```
#include<stdio.h>

#include<stdlib.h>

int t[100][100],v[100],w[100],n,m,i,j,count=0;

int max(int a,int b){

        return (a>b)?a:b;

}

int knap(int n,int m)

{

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

        {

                for(j=0;j<m+1;j++)
```

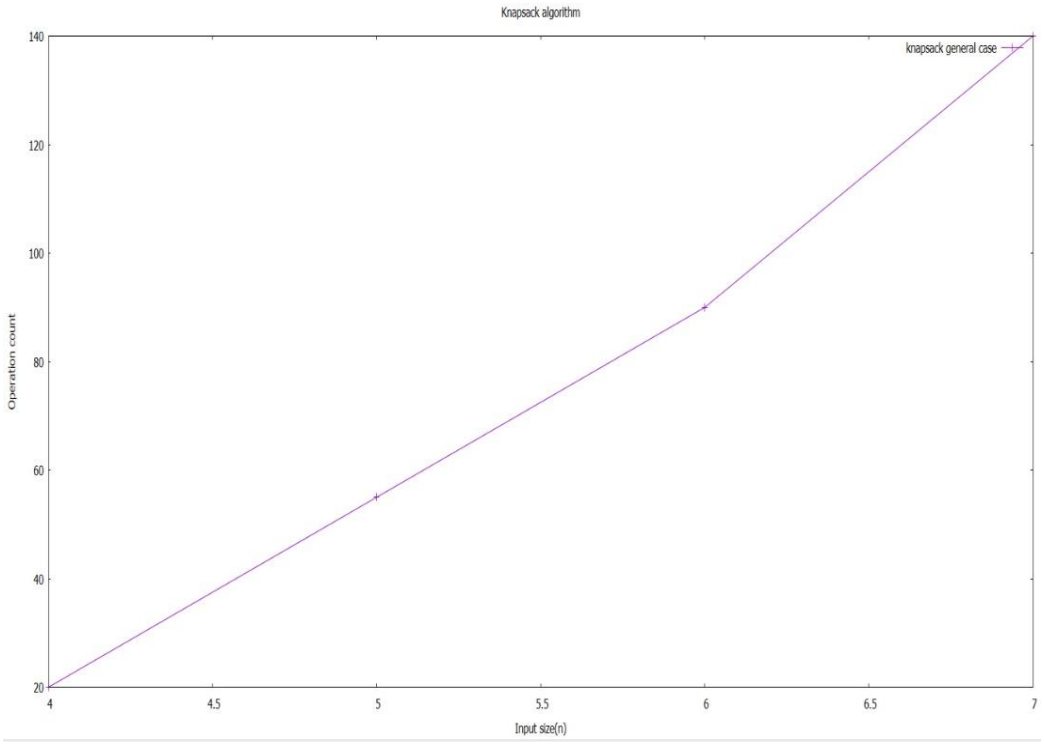
```
            {
                if(i==0||j==0)
                    t[i][j]=0;
                    else{
                    count++;
                if(j<w[i])
                    t[i][j]=t[i-1][j];
                else
                    t[i][j]=max(t[i-1][j],v[i]+t[i-1][j-w[i]]);
                    }
            }
        }
   return t[n][m];
}
void run()
{
        count=0;
        printf("Number of items:  ");
        scanf("%d",&n);
        printf("Sack capacity:  ");
```

```
scanf("%d",&m);
printf("Weight\tValue\n");
for(i=1;i<n+1;i++)
{
        scanf("%d\t%d",&w[i],&v[i]);
}
printf("Max value %d\n",knap(n,m));
for(int i=0;i<n+1;i++)
{
        for(int j=0;j<m+1;j++)
                printf("%d  ",t[i][j]);
        printf("\n");
}
printf("Composition:\n");
for(int i=n;i>0;i--)
{
        if(t[i][m]!=t[i-1][m])
        {
                printf("%d\t",i);
                m=m-w[i];
```

```
                }
        }
        printf("\n");
        printf("%d\t%d\n",n,count);
}
void main()
{
   FILE *f1;
   f1=fopen("knapsackgraph.txt","a");
   int ch;
   while(1)
   {
      printf("enter choice 1 to continue and 0 to exit\n");
      scanf("%d",&ch);
      switch(ch)
      {
            case 1:run();
                  fprintf(f1,"%d\t%d\n",n,count);
                  break;
            default:exit(0);
```

```
        }
    }
    fclose(f1);
}
```





Knapsack with memory function:

```
#include <stdio.h>

#include <stdlib.h>
```

```
int max(int a, int b){
	return (a>b) ? a : b;
}
int t[100][100], v[100], w[100], n, m, i, j;
int knap(int i, int j){
	if (t[i][j]==-1){
		if (j<w[i])
			t[i][j] = knap(i-1,j);
		else
			t[i][j] = max(knap(i-1,j),v[i]+knap(i-1,j-w[i]));
	}
	return t[i][j];
}
void main(){
	printf("No. of Items>> ");
	scanf("%d",&n);
	printf("Sack Capacity>> ");
	scanf("%d",&m);
	printf("Weight\tValue\n");
	for(i=1;i<n+1;i++){
```

```
        scanf("%d\t%d",&w[i],&v[i]);
    }

    for(i=0;i<n+1;i++){
        for(j=0;j<m+1;j++){
            if (i==0||j==0)
                t[i][j]=0;
            else
                t[i][j]=-1;
        }
    }

    printf("Maximum Value: %d\n",knap(n,m));
    for(int i=0;i<n+1;i++)
    {
        for(int j=0;j<m+1;j++)
            printf("%d  ",t[i][j]);
        printf("\n");
    }
    printf("Composition:\n");
```

```
        for(i=n;i>0;i--){
                if (t[i][m] != t[i-1][m]){
                        printf("%d ",i);
                        m = m-w[i];
                }
        }
        printf("\n");
}
```

```
No. of Items>> 5
Sack Capacity>> 11
Weight  Value
7       65
4       30
2       5
1       3
5       45
Maximum Value: 95
0       0       0       0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       65      65      65      65      65
0       -1      -1      0       30      30      30      -1      65      65      65      95
0       -1      -1      -1      -1      30      35      -1      -1      -1      70      95
0       -1      -1      -1      -1      -1      35      -1      -1      -1      -1      95
0       -1      -1      -1      -1      -1      -1      -1      -1      -1      -1      95
Composition:
2 1
```

b) Implement Prim's algorithm to find Minimum Spanning Tree of a graph and perform its analysis for different inputs.

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
int n, i, j, cost[10][10], cnt = 0, visited[10], removed[10];
int heapsize = 0;
```

```
struct edge
{
   int v;
   int dist;
   int u;
} heap[10],VT[10];
typedef struct edge edg;
// Min Heap function declaration
void swap(struct edge *a, struct edge *b)
{
   struct edge temp = *a;
   *a = *b;
   *b = temp;
}
void heapify(struct edge arr[], int n, int i)
{
   int largest = i;
   int left = 2 * i + 1;
   int right = 2 * i + 2;
   if (left < n && arr[left].dist < arr[largest].dist)
```

```
        largest = left;
    if (right < n && arr[right].dist < arr[largest].dist)
        largest = right;
    if (largest != i)
    {
        swap(&arr[i], &arr[largest]);
        heapify(arr, n, largest);
    }
}
void heapSort(struct edge arr[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
    {
        heapify(arr, n, i);
    }
}
// Min heap function declaration end

void makegraph()
{
```

```
    // Make Graph
    printf("Enter the total number of vertices:");
    scanf("%d", &n);
    printf("Enter the cost matrix of the Graph\n");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            scanf("%d", &cost[i][j]);
            if (cost[i][j] == 0)
                cost[i][j] = INT_MAX;
        }
    }
}
// returns the min of the heap //astey
edg deleteheap(edg heap[])
{
    edg min = heap[0];
    heap[0] = heap[heapsize - 1];
    heapsize = heapsize - 1;
```

```
    return min;
}
void prim()
{
    // Appending Souce vertex to heap and incrementing heap size
    visited[0] = 1;
    heap[heapsize].v = -1;
    heap[heapsize].u = 0;
    heap[heapsize].dist = 0;
    heapsize++;
    while (cnt != n)
    {
        // fetching the min and appending to the visited array of edges and deleting
from heap
        edg min = deleteheap(heap);
        VT[cnt].v = min.v;
        VT[cnt].u = min.u;
        VT[cnt].dist = min.dist;
        cnt++;
        int v = min.u;
        removed[v] = 1;
```

```
for (i = 1; i < n; i++)
{
   if (!visited[i] && cost[v][i] != INT_MAX && !removed[i])
   {
      // not visited and not removed from heap
      visited[i] = 1;
      heap[heapsize].v = v;
      heap[heapsize].u = i;
      heap[heapsize].dist = cost[v][i];
      heapsize++;
   }
   if (visited[i] && cost[v][i] != INT_MAX && !removed[i])
   { // visited but not removed from heap --> scope for minimisation?
      for (j = 0; j < heapsize; j++)
      { // finding that edge in the sorted heap
         if (heap[j].u == i && cost[v][i] < heap[j].dist)
         { // replacing if optimal
            heap[j].dist = cost[v][i];
            heap[j].v = v;
            break;
```

```
                }
            }
        }
    }
    heapSort(heap, heapsize); // sorting after deletions and value modifications
  }
}
void main()
{
  int sum = 0;
  makegraph();
  prim();
  for (int i = 1; i < cnt; i++)
  {
    printf("%c --> %c == %d\n", VT[i].v + 65, VT[i].u + 65, VT[i].dist);
    sum += VT[i].dist;
  }
  printf("Minimum Distance is: %d", sum);
}
```

```
Enter the total number of vertices:4
Enter the cost matrix of the Graph
4 0 1 0
4 0 6 2
1 6 0 3
0 2 3 0
A --> C == 1
C --> D == 3
D --> B == 2
Minimum Distance is: 6
```

PLOTTER:

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
int n, i, j, cost[10][10], cnt = 0, visited[10], removed[10];
int heapsize = 0;
int heapcount,graphcount,max;
struct edge
{
   int v;
   int dist;
   int u;
} heap[10],VT[10];
typedef struct edge edg;
// Min Heap function declaration
void swap(struct edge *a, struct edge *b)
{
   struct edge temp = *a;
   *a = *b;
   *b = temp;
}
```

```
void heapify(struct edge arr[], int n, int i)
{
   int largest = i;
   int left = 2 * i + 1;
   int right = 2 * i + 2;
   heapcount++;
   if (left < n && arr[left].dist < arr[largest].dist)
      largest = left;
   if (right < n && arr[right].dist < arr[largest].dist)
      largest = right;

   if (largest != i)
   {
      swap(&arr[i], &arr[largest]);
      heapify(arr, n, largest);
   }
}
void heapSort(struct edge arr[], int n)
{
   for (int i = n / 2 - 1; i >= 0; i--)
   {
      heapify(arr, n, i);
   }
}
// Min heap function declaration end

void makegraph()
```

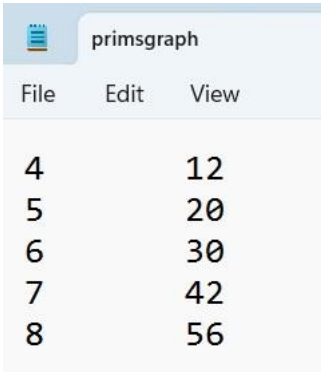
```
{
   // Make Graph
   printf("Enter the total number of vertices:");
   scanf("%d", &n);
   printf("Enter the cost matrix of the Graph\n");
   for (i = 0; i < n; i++)
   {
      for (j = 0; j < n; j++)
      {
         scanf("%d", &cost[i][j]);
         if (cost[i][j] == 0)
            cost[i][j] = INT_MAX;
      }
   }
}
// returns the min of the heap //astey
edg deleteheap(edg heap[])
{
   edg min = heap[0];
   heap[0] = heap[heapsize - 1];
   heapsize = heapsize - 1;
   return min;
}
void prim()
{
   // Appending Souce vertex to heap and incrementing heap size
   visited[0] = 1;
```

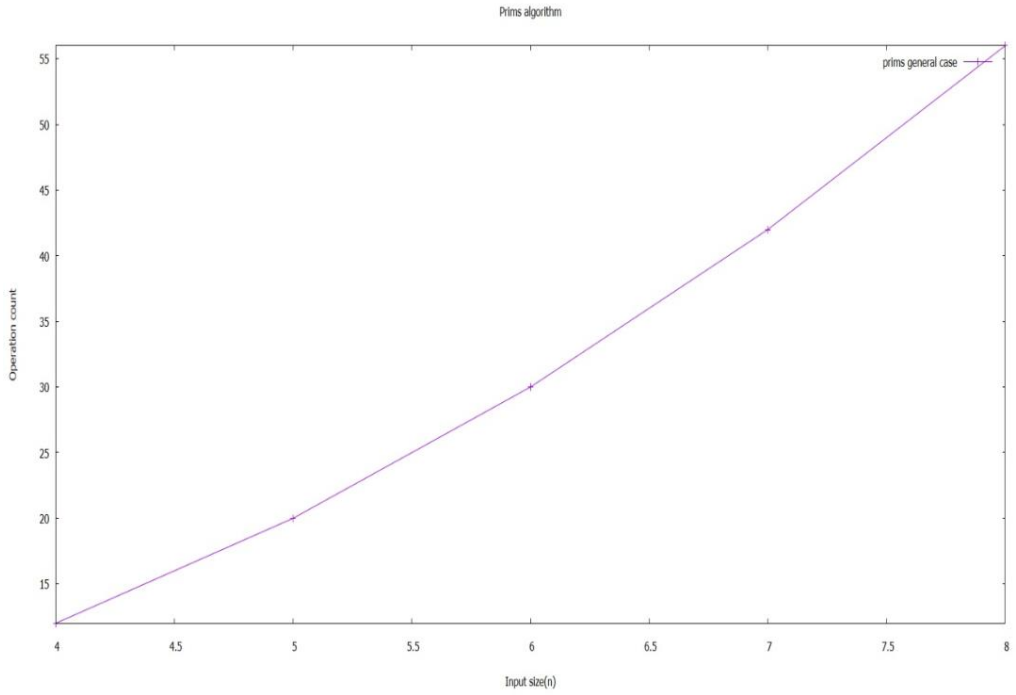
```
    heap[heapsize].v = -1;
    heap[heapsize].u = 0;
    heap[heapsize].dist = 0;
    heapsize++;

    while (cnt != n)
    {
        // fetching the min and appending to the visited array of edges and deleting
from heap
        edg min = deleteheap(heap);
        VT[cnt].v = min.v;
        VT[cnt].u = min.u;
        VT[cnt].dist = min.dist;
        cnt++;
        int v = min.u;
        removed[v] = 1;
        for (i = 1; i < n; i++)
        {
            graphcount++;
            if (!visited[i] && cost[v][i] != INT_MAX && !removed[i])
            {
                // not visited and not removed from heap
                visited[i] = 1;
                heap[heapsize].v = v;
                heap[heapsize].u = i;
                heap[heapsize].dist = cost[v][i];
                heapsize++;
            }
```

```
            if (visited[i] && cost[v][i] != INT_MAX && !removed[i])
            { // visited but not removed from heap --> scope for minimisation?
               //graphcount++;
               for (j = 0; j < heapsize; j++)
               { // finding that edge in the sorted heap
                  if (heap[j].u == i && cost[v][i] < heap[j].dist)
                  { // replacing if optimal
                     heap[j].dist = cost[v][i];
                     heap[j].v = v;
                     break;
                  }
               }
            }
         }
         heapSort(heap, heapsize); // sorting after deletions and value modifications
      }
   }
   void run()
   {
      int sum = 0;cnt = 0;heapsize = 0;heapcount=0;graphcount=0;max=0;
      makegraph();
      prim();
      for (int i = 1; i < cnt; i++)
      {
         printf("%c --> %c == %d\n", VT[i].v + 65, VT[i].u + 65, VT[i].dist);
         sum += VT[i].dist;
      }
```

```
    printf("Minimum Distance is: %d", sum);
    max=(graphcount>heapcount)?graphcount:heapcount;
    printf("basic count=%d",max);
}

void main()
{

    FILE *f1;
    f1=fopen("primsgraph.txt","a");
    int ch;
    while(1)
    {

        printf("enter choice 1 to continue and 0 to exit\n");
        scanf("%d",&ch);
        switch(ch)
        {

            case 1:run();
                    fprintf(f1,"%d\t%d\n",n,max);
                    break;
            default:exit(0);

        }
    }
    fclose(f1);
```

}





12. Implement Dijkstra's algorithm to find shortest paths to other vertices in a graph and perform its analysis.

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
int n, i, j, src, cost[10][10], d[10] = {0}, removed[10] = {0}, count = 0;
int heapsize;
struct vertex
```

```
{
    int id;
    int dist;
} heap[10];
typedef struct vertex ver;

// Min Heap function declaration
void swap(struct vertex *a, struct vertex *b)
{
    struct vertex temp = *a;
    *a = *b;
    *b = temp;
}
void heapify(struct vertex arr[], int n, int i)
{
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left].dist < arr[largest].dist)
        largest = left;
    if (right < n && arr[right].dist < arr[largest].dist)
        largest = right;

    if (largest != i)
    {
        swap(&arr[i], &arr[largest]);
```

```
        heapify(arr, n, largest);
    }
}
void heapSort(struct vertex arr[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
    {
        heapify(arr, n, i);
    }
}
// Min heap function declaration end

void makegraph()
{
    // Make Graph
    printf("Enter the total number of vertices:");
    scanf("%d", &n);
    printf("Enter the cost matrix of the Graph\n");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            scanf("%d", &cost[i][j]);
            if (cost[i][j] == 0)
                cost[i][j] = INT_MAX;
        }
    }
```

```
    // Initialise the source vertex distance to 0 and rest all to infinity(INT_MAX)
    printf("Enter the source vertex:");
    scanf("%d", &src);
    for (i = 0; i < n; i++)
    {
        d[i] = INT_MAX;
    }
    d[src] = 0;
}
// returns the min of the heap and heapifies the rest of the elements
ver deleteheap(ver heap[])
{
    ver min = heap[0];
    heap[0] = heap[heapsize - 1];
    heapsize = heapsize - 1;
    heapify(heap, heapsize, 0);
    return min;
}
void dijkstra()
{
    for (i = 0; i < n; i++)
    {
        heap[i].id = i;
        heap[i].dist = INT_MAX;
    }
    heap[src].dist = 0;
```

```
heapsize = n;
// pulling source to index 0
heapSort(heap, heapsize);
while (count < n)
{
   ver minvertex = deleteheap(heap);
   int u = minvertex.id;
   removed[u] = 1;
   count++;
     for (i = 0; i < n; i++)
   {
      if (!removed[i] && cost[u][i] != INT_MAX)
      {
         if ((d[u] + cost[u][i]) < d[i])
         {
            d[i] = (d[u] + cost[u][i]);
            for (int o = 0; o < heapsize; o++)
            {
               if (heap[o].id == i)
               {
                  heap[o].dist = d[i];
                  break;
               }
            }
            // to sort after editing
            heapSort(heap, heapsize);
         }
```

```
        }
      }
    }
}
void main()
{
   makegraph();
   dijkstra();
   printf("Shortest path id %d is:\n", src);
   for (i = 0; i < n; i++){
      if (src != i)
         printf("%d -> %d = %d\n", src, i, d[i]);
   }
}
```

```
Enter the total number of vertices:5
Enter the cost matrix of the Graph
0 5 0 6 0
5 0 1 3 0
0 1 0 4 6
6 3 4 0 2
0 0 6 2 0
Enter the source vertex:a
Shortest path id 0 is:
0 -> 1 = 5
0 -> 2 = 6
0 -> 3 = 6
0 -> 4 = 8
```

PLOTTER:

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
int n, i, j, src, cost[10][10], d[10] = {0}, removed[10] = {0}, count = 0;
int heapsize;int graphcount,heapcount,max;
```

```
struct vertex
{
   int id;
   int dist;
} heap[10];
typedef struct vertex ver;

// Min Heap function declaration
void swap(struct vertex *a, struct vertex *b)
{
   struct vertex temp = *a;
   *a = *b;
   *b = temp;
}
void heapify(struct vertex arr[], int n, int i)
{
   int largest = i;
   int left = 2 * i + 1;
   int right = 2 * i + 2;
   heapcount++;
   if (left < n && arr[left].dist < arr[largest].dist)
      largest = left;
   if (right < n && arr[right].dist < arr[largest].dist)
      largest = right;
    if (largest != i)
   {
      swap(&arr[i], &arr[largest]);
```

```
        heapify(arr, n, largest);
    }
}
void heapSort(struct vertex arr[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
    {
        heapify(arr, n, i);
    }
}
// Min heap function declaration end
void makegraph()
{
    // Make Graph
    printf("Enter the total number of vertices:");
    scanf("%d", &n);
    printf("Enter the cost matrix of the Graph\n");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            scanf("%d", &cost[i][j]);
            if (cost[i][j] == 0)
                cost[i][j] = INT_MAX;
        }
    }
   // Initialise the source vertex distance to 0 and rest all to infinity(INT_MAX)
```

```
   printf("Enter the source vertex:");
    scanf("%d", &src);
    for (i = 0; i < n; i++)
    {
        d[i] = INT_MAX;
    }
    d[src] = 0;
}
// returns the min of the heap and heapifies the rest of the elements
ver deleteheap(ver heap[])
{
    ver min = heap[0];
    heap[0] = heap[heapsize - 1];
    heapsize = heapsize - 1;
    heapify(heap, heapsize, 0);
    return min;
}
void dijkstra()
{
    for (i = 0; i < n; i++)
    {
        heap[i].id = i;
        heap[i].dist = INT_MAX;
    }
    heap[src].dist = 0;
    heapsize = n;
    // pulling source to index 0
```

```
heapSort(heap, heapsize);
while (count < n)
{
    ver minvertex = deleteheap(heap);
    int u = minvertex.id;
    removed[u] = 1;
    count++;
   for (i = 0; i < n; i++)
    {
        if (!removed[i] && cost[u][i] != INT_MAX)
        {
            graphcount++;
            if ((d[u] + cost[u][i]) < d[i])
            {
                d[i] = (d[u] + cost[u][i]);
                for (int o = 0; o < heapsize; o++)
                {
                    if (heap[o].id == i)
                    {
                        heap[o].dist = d[i];
                        break;
                    }
                }
                // to sort after editing
                heapSort(heap, heapsize);
            }
        }
```

```
        }
    }
}
void run()
{
    makegraph();max=0;graphcount=0;heapcount=0;count=0;
    dijkstra();
    printf("Shortest path id %d is:\n", src);
    for (i = 0; i < n; i++)
    {
        if (src != i)
            printf("%d -> %d = %d\n", src, i, d[i]);
    }
    max=(graphcount>heapcount)?graphcount:heapcount;
    printf("basic count=%d",max);
}
void main()
{
    FILE *f1;
    f1=fopen("dijkstrasgraph.txt","a");
    int ch;
    while(1)
    {
        printf("enter choice 1 to continue and 0 to exit\n");
        scanf("%d",&ch);
        switch(ch)
        {
```

```
        case 1:run();
                fprintf(f1,"%d\t%d\n",n,max);
                break;
         default:exit(0);

      }
   }
   fclose(f1);
}
```