ADA LAB PROGRAMS

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.

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

int modifiedeuclid(int m, int n)
{
    int r;
    int count = 0;
    while (n)
    {
        count++;
        r = m % n;
        m = n;
        n = r;
    }
    return count;//m is gcd
}

int consec(int m, int n)
{
    int min=m;
    int count = 0;
    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;//min is gcd
}

int euclid(int m, int n)
{
    int temp;
    int count = 0;
    while (n > 0)
    {
        count++;
        if (n > m)
        {
            temp = m;
            m = n;
            n = temp;
        }
        m = m - n;
    }
    return count; // m is the GCD
}

void analysis(int ch)
{
```

```
int m, n, i, j, k,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 data
    {
        for (k = 2; k <= i; k++)
        {
            count = 0;
            m = j;
            n = k;

            switch (ch)
            {
            case 1: count = modifiedeuclid(m, n);
                    break;
            case 2: count = consec(m, n);
                    break;
            case 3: count = euclid(m, n);
                    break;
            }

            // To find maximum basic operations among all
combinations between 2 to n
            if (count > maxcount)
                maxcount = count;
            // To find minimum basic operations among all
combinations between 2 to n
            if (count < mincount)
                mincount = count;
        }
    }
    switch (ch)
```

```
        {
        case 1: fp1 = fopen("me_b.txt", "a");
                fp2 = fopen("me_w.txt", "a");
                break;
        case 2: fp1 = fopen("c_b.txt", "a");
                fp2 = fopen("c_w.txt", "a");
                break;
        case 3: fp1 = fopen("e_b.txt", "a");
                fp2 = fopen("e_w.txt", "a");
                break;
        }
        fprintf(fp1, "%d\t%d\n", i, mincount);
        fclose(fp1);
        fprintf(fp2, "%d\t%d\n", i, maxcount);
        fclose(fp2);
    }
}

void main()
{
    analysis(1);
    analysis(2);
    analysis(3);

    system("gnuplot>load 'gcd_plot.txt' ");
}
```

```
set title 'GCD Plot'
set xrange [0:100]
set yrange [0:150]
set xlabel 'Input'
set ylabel 'Count'
set style data linespoints
plot 'me_b.txt' u 1:2 w lp,'me_w.txt' u 1:2 w lp,'c_b.txt' u 1:2 w lp,'c_w.txt' u 1:2 w lp,'e_b.txt' u 1:2 w lp,'e_w.txt' u 1:2 w lp
```

pause-1 "Hit any Key to continue"

2 Implement the following searching algorithms and perform their analysis by generating best case and worst case data.

a) Sequential search

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

int linearSearch(int arr[], int val, int length)
{
  int count = 0, found = 0;
  for (int i = 0; i < length; i++)
  {
    count++;
    if (arr[i] == val)
    {
      // printf("Element %d found at index %d\n",val,i);
      return count;
    }
  }
  // printf("Element %d not found\n",val);
  return count;
}

void main()
{
  srand(time(NULL));
  int x, size = 1, key = 0;
  FILE *fptr = fopen("timingLinear.txt", "w");
  while (size < 40000)
```

```c
  {
    if (size < 10000) size *= 10;
    else size *= 2;

    int *arr = (int *)malloc(sizeof(int) * size);
    for (int i = 0; i < size; i++)
    {
      arr[i] = rand();
    }

    // BEST CASE
    x = linearSearch(arr, arr[0], size);
    // printf("BEST CASE: Linear search took %d counts \n", x);
    fprintf(fptr, "%d\t%d\t", size, x);

    // AVG CASE
    x = linearSearch(arr, arr[size / 2], size);
    // printf("AVG CASE: Binary search took %d \n", x);
    fprintf(fptr, "%d\t%d\t", size, x);

    // WORST CASE
    x = linearSearch(arr, arr[size - 1], size);
    // printf("WORST CASE: Linear search took %d counts \n", x);
    fprintf(fptr, "%d\t%d\n", size, x);
  }
  fclose(fptr);
}
```

```
set title "Linear Search"
set xrange [0:100]
set yrange [0:30]
set xlabel "Number of elements(n)"
set ylabel "Basic Operation Count"
set style data linespoints
```

```
plot 'timingLinear.txt' u 1:2 w lp, 'timingLinear.txt' u 3:4 w lp,
'timingLinear.txt' u 5:6 w lp
```

b) Binary search (recursive)

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

int binarySearch(int arr[], int low, int high, int x, int *count) {
   if (low > high) {
      *count += 1; // element not found
      return -1;
   }
   int mid = (low + high) / 2;
   if (arr[mid] == x) {
      *count += 2; // element found at mid index
      return mid;
   }
   else if (arr[mid] > x) {
      *count += 3; // compare, then search left subarray
      return binarySearch(arr, low, mid-1, x, count);
   }
   else {
      *count += 3; // compare, then search right subarray
      return binarySearch(arr, mid+1, high, x, count);
   }
}

void main() {
   int values[] = {1000, 2000, 5000, 10000, 20000, 30000, 40000,
50000};
   int numOfValues = sizeof(values) / sizeof(int);
```

```c
    FILE *fptr = fopen("count.dat", "w");

    for (int i = 0; i < numOfValues; i++) {
        int n = values[i];
        int arr[n];
        for (int j = 0; j < n; j++) {
            arr[j] = j;
        }

        // Best case: element is present at mid index
        int x = arr[(n-1)/2];
        int count = 0;
        binarySearch(arr, 0, n-1, x, &count);
        fprintf(fptr, "%d\t%d\t",n, count);

        // Worst case: element is not present in array
        x = -1;
        count = 0;
        binarySearch(arr, 0, n-1, x, &count);
        fprintf(fptr, "%d\t%d\n",n, count);
    }

    fclose(fptr);
}
```

```
set title "Recursive Binary Search"
set xrange [1000:50000]
set yrange [1: 50]
set xlabel 'Number of elemnts(n)'
set ylabel 'Count(Number of Operations)'
set style data linespoints
plot 'count.dat' u 1:2 w lp ,'count.dat' u 3:4 w lp
```

3.Implement the following elementary sorting algorithms and perform their analysis by
generating best case and worst case data. (Note: Any two may be asked in the
test/exam)

a) Selection sort

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

int SelectionSort(int a[], int n)
{
    int count = 0, min, temp;
    for (int i = 0; i < n - 1; i++)
    {
        min = i;
        for (int j = i + 1; j < n; j++)
        {
            count++;
            if (a[j] < a[min]) min = j;
        }
        temp = a[i];
        a[i] = a[min];
        a[min] = temp;
    }
    return count;
}

void main()
{
    int n = 1, count;
    FILE *fptr=fopen("data.txt", "w");
```

```
    while (n < 40000)
    {
        if (n < 10000) n *= 10;
        else n *= 2;
        int *a = (int *)malloc(sizeof(int) * n);
        for (int i = 0; i < n; i++)
        {
            a[i] = i;
        }
        count = SelectionSort(a, n);
        fprintf(fptr, "%d\t%d\n", n, count);
    }
    fclose(fptr);
    system("gnuplot>load 'selection_plot.txt' ");
}
```

```
set title "Selection Sort"
set xrange [10:40000]
set xlabel 'Number of elements(n)'
set ylabel 'Count'
set style data linespoints
plot 'data.txt' u 1:2 w lp
pause -1 "Hit any Key to Continue"
```

b)Bubble sort

```
#include <stdio.h>
#include <stdlib.h>
int bubble(int *a, int n)
{
```

```
    int c = 0, i, j, t, f = 0;
    for (i = 0; i < n - 1; i++)
    {
        f = 0;
        for (j = 0; j < n - i - 1; j++)
        {
            c++;
            if (a[j] > a[j + 1])
            {
                f = 1;
                t = *(a + j);
                *(a + j) = *(a + j + 1);
                *(a + j + 1) = t;
            }
        }
        if (f == 0)
            return c;
    }
    return c;
}
void main()
{
    int ct, i = 100, k, *a;
    FILE *f;
    ct = 0;
    f = fopen("bub_time.txt", "a");

    while (i <= 40000)
    {
        a = (int *)malloc(sizeof(int) * i);
        // Best case
        for (k = 1; k <= i; k++)
            *(a + k) = k;
        ct = bubble(a, i);
        printf("bestfor i: %d\tct :%d\n", i, ct);
```

```c
        fprintf(f, "%d\t\t%d\t\t", i, ct);
        // Worst Case
        for (k = 1; k <= i; k++)
            *(a + k) = i - k;
        ct = bubble(a, i);
        printf("worstfor i: %d\tct :%d\n", i, ct);
        fprintf(f, "%d\t\t%d\n", i, ct);
        if (i < 10000)
            i *= 10;
        else
            i *= 2;
    }
    fclose(f);
    system("gnuplot >load 'bubble_plot1.txt'");
    system("gnuplot >load 'bubble_plot2.txt'");
}
```

```
set title 'Bubble Plot'
set xrange [0:40000]
set yrange [0:40000]
set xlabel 'Input Size '
set ylabel 'Basic Operation Count'
set style data linespoints
plot 'bub_time.txt' u 1:2 w lp
pause -1 "Hit any key to continue"

set title 'Bubble Plot'
set xrange [0:40000]
set xlabel 'Input Size'
set ylabel 'Basic Opereation Count'
set style data linespoints
plot 'bub_time.txt' u 3:4 w lp
pause -1 "Hit any key to continue"
```

c)Insertion sort

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

void main()
{
  int *arr, i, n = 1, count;

  srand(time(NULL));
```

```
FILE *f;

f = fopen("insertion_time.txt", "w");

while (n < 40000)
{
  if (n < 10000)
    n *= 10;
  else
    n *= 2;

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

  // best case
  count = 0;
  for (i = 0; i < n; i++)
    *(arr + i) = i + 1;
  insertionSort(arr, n, &count);
  fprintf(f, "%d\t%d\t", n, count);

  // avg case
  count = 0;
  for (i = 0; i < n; i++)
    *(arr + i) = rand() % n;
  insertionSort(arr, n, &count);
  fprintf(f, "%d\t%d\t", n, count);

  // worst case
  count = 0;
  for (i = 0; i < n; i++)
    *(arr + i) = n - i;
  insertionSort(arr, n, &count);
  fprintf(f, "%d\t%d\n", n, count);
```

```
    free(arr);
  }
  fclose(f);
  // system("gnuplot>load 'insertion_plot.txt'");
}
```

```
set title 'Insertion Sort'
set xrange [0:50000]
set yrange [-2:900000000]
set xlabel 'Input'
set ylabel 'Count'
set style data linespoints
plot 'insertion_time.txt' u 1:2 w lp,'insertion_time.txt' u 3:4 w lp,
'insertion_time.txt' u 5:6 w lp

pause -1 "Hit any Key to Continue"
```

4 Implement Brute force string matching algorithm to search for a pattern of length ‘M’ in a text of length ‘N’ (M<=N) and perform its analysis by generating best case and worst case data.

```
#include <stdio.h>
#include <stdlib.h>
int stringMatching(char T[], char P[], int M, int N)
{
    int count = 0;
    for (int i = 0; i <= N - M; i++)
    {
        int j = 0;
        while ((j < M) && (P[j] == T[i + j]))
```

```
        {
            count++;
            j++;
        }
        if (j == M)
            return count;
        count++;
    }
    return count;
}
void main()
{
    int count1, count2, N = 1000, M = 1;
    char T[N];
    FILE *fp;
    fp = fopen("StringMatch.txt", "a");
    for (int i = 0; i < N; i++)
    {
        T[i] = 'A';
    }
    while (M < N)
    {
        if (M == 1)
            M *= 100;
        else
            M += 100;
        char P[M];
        for (int i = 0; i < M; i++)
        {
            P[i] = 'A';
        }
        count1 = stringMatching(T, P, M, N);
        fprintf(fp, "%d\t %d\t\t", M, count1);
        P[M - 1] = 'B';
        count2 = stringMatching(T, P, M, N);
```

```
        fprintf(fp, "%d\t%d\t\n", M, count2);
    }
    fclose(fp);
    system("gnuplot > load 'string_plot1.txt'");
    system("gnuplot > load 'string_plot2.txt'");
}
```

```
set title 'Best Case String Match Plot'
set xrange [0:1000]
set yrange [0:1000]
set xlabel 'Input Size'
set ylabel 'Basic Operation count'
set style data linespoints
plot 'StringMatch.txt' u 1:2 w lp
pause -1 "Hit any key to continue"

set title 'Best Case String Match Plot'
set xrange [0:1000]
set yrange [0:300000]
set xlabel 'Input Size'
set ylabel 'Basic Operation count'
set style data linespoints
plot 'StringMatch.txt' u 3:4 w lp
pause -1 "Hit any key to continue"
```

5 Implement Merge Sort algorithm and perform its analysis by generating best case and worst case data

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

```
#include <stdlib.h>
#include <time.h>
void worst_merge(int a[], int lf[], int rt[], int l, int m, int r)
{
    int i;
    for (i = 0; i <= m - l; i++)
    {
        a[i] = lf[i];
    }
    for (int j = 0; j < r - m; j++)
    {
        a[i + j] = rt[j];
    }
}
void split(int a[], int lf[], int rt[], int l, int m, int r)
{
    int i;
    for (i = 0; i <= m - l; i++)
    {
        lf[i] = a[i * 2];
    }
    for (i = 0; i < r - m; i++)
    {
        rt[i] = a[i * 2 + 1];
    }
}
void gen_worst_data(int a[], int l, int r)
{
    if (l < r)
    {
        int m = l + (r - l) / 2;
        int lf[m - l + 1];
        int rt[r - m];
        split(a, lf, rt, l, m, r);
        gen_worst_data(lf, l, m);
```

```
        gen_worst_data(rt, m + 1, r);
        worst_merge(a, lf, rt, l, m, r);
    }
}
int merge(int b[], int c[], int a[], int n, int m)
{
    int i = 0, j = 0, k = 0, ct = 0;
    while (i < n && j < m)   //blen=n,clen=m
    {
        if (b[i] < c[j])
        {
            a[k++] = b[i++];
            //i++;
        }
        else
        {
            a[k++] = c[j++];
            //j++;
        }
        //k++;
        ct++;//*Increment count only here
    }
    while (i < n)
    {
        a[k++] = b[i++];
       // i++;
        //k++;
    }
    while (j < m)
    {
        a[k++] = c[j++];
        //j++;
        //k++;
    }
    return ct;
```

```
}
void mergeSort(int a[], int n, int *ct)
{
    int i = 0, j = 0;
    if (n == 1)
        return;
    else
    {
        int b[n / 2];
        int c[n / 2];
        int i, blen = 0, clen = 0;
        for (i = 0; i < n / 2; i++)
        {
            b[i] = a[i];
            blen++;
        }
        i = 0;
        for (j = n / 2; j < n; j++)
        {
            c[i] = a[j];
            i++;
            clen++;
        }
        mergeSort(b, n / 2, ct);
        mergeSort(c, n / 2, ct);
        *(ct) += merge(b, c, a, blen, clen);
    }
}
void main()
{
    srand(time(NULL));
    FILE *f;
    int n = 4, *a, i, ct = 0;
    f = fopen("mergect.txt", "w");
    while (n <= 4096)
```

```c
    {
        n *= 2;
        ct = 0;
        a = (int *)malloc(n * sizeof(int));
        // best case
        for (i = 0; i < n; i++)
            a[i] = i + 1;
        mergeSort(a, n, &ct);
        printf("best ct %d\t for n: %d\t", ct, n);
        fprintf(f, "%d\t%d\t", n, ct);
        // Average case
        ct = 0;
        for (i = 0; i < n; i++)
            a[i] = rand();
        mergeSort(a, n, &ct);
        printf("average ct %d\t for n: %d\t", ct, n);
        fprintf(f, "%d\t%d\t", n, ct);
        // worst case
        ct = 0;
        gen_worst_data(a, 0, n - 1);
        mergeSort(a, n, &ct);
        printf("worst ct %d\t for n: %d\n", ct, n);
        fprintf(f, "%d\t%d\n", n, ct);
    }
    fclose(f);
    system("gnuplot > load 'merge_plot.txt' ");
}
```

```
set title 'Merge Plot'
set xrange [8:9000]
set yrange [10:160000]
set xlabel 'Input'
set ylabel 'Count'
set style data linespoints
```

plot 'mergect.txt' u 1:2 w lp,'mergect.txt' u 3:4 w lp,'mergect.txt' u 5:6 w lp

6 Implement Quick Sort algorithm and perform its by generating best case and worst case data

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

void swap(int *a,int *b)
 {
  int temp=*a;
  *a=*b;
  *b=temp;
 }

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

void main()
{

 int *arr,n,count;

 srand(time(NULL));
 FILE *f;

 f=fopen("quickcount.txt","w");

 n=4;
 while(n<1034)
 {
  arr=(int *)malloc(sizeof(int)*n);

  //Best case All same elements
  count=0;
  for(int i=0;i<n;i++)
      *(arr+i)=5;
```

```
 quicksort(arr,0,n-1,&count);
 fprintf(f,"%d\t%d\t",n,count);
 printf("%d\t%d\t",n,count);



 //AVG case Random input
 for(int i=0;i<n;i++)
    *(arr+i)=rand()%n;
 count=0;
 quicksort(arr,0,n-1,&count);
 fprintf(f,"%d\t%d\t",n,count);
 printf("%d\t%d\t",n,count);

 //worst case Decreasing input
 count=0;
 for(int i=0;i<n;i++)
   *(arr+i)=n-i;
 quicksort(arr,0,n-1,&count);
 fprintf(f,"%d\t%d\n",n,count);
 printf("%d\t%d\n",n,count);

 n=n*2;

free(arr);
 }

 fclose(f);

}

set title 'Quick  Plot'
set xrange [0:1500]
set yrange [10:12000]
set xlabel 'Input'
```

```
set ylabel 'Count'
set style data linespoints
plot 'quickcount.txt' u 1:2 w lp,'quickcount.txt' u 3:4 w
lp,'quickcount.txt' u 5:6 w lp
```

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.

```
#include <stdio.h>
#include <stdlib.h>
int n, count = 0;
int ct = 0, k = 0;
void genData(int a[][n], int size, int ch) // ch=0 for worst,ch=1 for
best
{
    printf("Graph is\n");
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            if (i == j && ch == 0)
                a[i][j] = 0;
            else if (ch == 0)
                a[i][j] = 1;
            else if (j - 1 == i && ch == 1)
                a[i][j] = 1;
            else
                a[i][j] = 0;
            printf("%d\t", a[i][j]);
```

```
        }
        printf("\n");
    }
    printf("\n");
}
void dfs(int a[][n], int i, int s, int visited[], int *c, int comp[], int *cycfg,
int p)
{
    if (k == 0 && i != s)
    {
        visited[i] = (*c) * n;
        k = *c;
    }
    else
    {
        *(c) += 1;
        visited[i] = *c;
    }
    printf("%c\t", i + 65);
    comp[k] = i;
    k += 1;
    for (int j = 0; j < n; j++)
    {
        count++;
        if (visited[j] == 0 && a[i][j] == 1)
            dfs(a, j, s, visited, c, comp, cycfg, i);
        else
        {
            if (a[i][j] == 1 && i != p)
                *(cycfg) = 1;
        }
    }
}
void dfs_help(int a[][n], int s, int visited[], int *c, int comp[], int *cycfg)
{
```

```
    for (int i = 0; i < n; i++)
    {
        visited[i] = 0;
        comp[i] = 0;
    }
    for (int i = s; (i % (n + 1)) < n; i++)
    {
        i = i % (n + 1);
        if (visited[i] == 0)
        {
            ct += 1;
            k = 0;
            dfs(a, i, s, visited, c, comp, cycfg, -1);
        }
    }
}
void printComponent(int comp[], int visited[], int n)
{
    int i = 0, p = 1;
    while (i < n)
    {
        printf("Component %d is\n", p);
        p++;
        do
        {
            printf("%c\t", comp[i] + 65);
            i++;
        } while (visited[i] <= n && i < n);
        printf("\n");
    }
}
void main()
{
    FILE *f = fopen("dfs_count.txt", "w");
    int c = 0, s, p = 1, cycfg = 0;
```

```c
char m;
n = 4;
while (n <= 6)
{
    count = 0, c = 0, p = 1, ct = 0, k = 0, cycfg = 0;
    int a[n][n], b[n][n];
    int visited[n];
    int comp[n];

    // best case
    genData(a, n, 1);
    printf("enter start node.Nodes are A,B...\n");
    scanf(" %c", &m);
    s = m - 65;
    printf("Traversal\n");
    dfs_help(a, s, visited, &c, comp, &cycfg);
    printf("\n");
    if (cycfg == 1)
        printf("Cyclic\n");
    else
        printf("Acyclic\n");
    if (ct > 1)
        printf("Disconnected\n");
    else
        printf("Connected\n");
    printf("Connected components are\n");
    printComponent(comp, visited, n);
    printf("Count : %d", count);
    fprintf(f, "%d\t%d\n", n, count);
    printf("\n");

    // worst case
    count = 0, c = 0, p = 1, ct = 0, k = 0, cycfg = 0;
    genData(b, n, 0);
    printf("enter start node.Nodes are A,B...\n");
```

```
        scanf(" %c", &m);
        s = m - 65;
        printf("Traversal\n");
        dfs_help(b, s, visited, &c, comp, &cycfg);
        printf("\n");
        if (cycfg == 1)
            printf("Cyclic\n");
        else
            printf("Acyclic\n");
        if (ct > 1)
            printf("Disconnected\n");
        else
            printf("Connected\n");
        printf("Connected components are\n");
        printComponent(comp, visited, n);
        printf("Count : %d", count);
        fprintf(f, "%d\t%d\n", n, count);
        printf("\n");
        n += 1;
    }
}
```

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

```
#include <stdio.h>
#include <stdlib.h>
int n, count = 0;
int ct = 1, k = 0;
struct q
{
```

```
    int *arr, f, r, c;
};
void init(struct q *qu, int n)
{
    qu->arr = malloc(n * sizeof(int));
    qu->f = 0;
    qu->r = -1;
    qu->c = 0;
}
void insert(struct q *qu, int k)
{
    qu->arr[++qu->r] = k;
    qu->c++;
}
void del(struct q *qu)
{
    qu->f++;
    qu->c--;
}
int isEmpty(struct q *qu)
{
    if (qu->c == 0)
        return 1;
    else
        return 0;
}
void genData(int a[][n], int size, int ch) // ch=0 for worst,ch=1 for
best
{
    printf("Graph is\n");
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            if (i == j && ch == 0)
```

```
            a[i][j] = 0;
        else if (ch == 0)
            a[i][j] = 1;
        else if (j - 1 == i && ch == 1)
            a[i][j] = 1;
        else
            a[i][j] = 0;
        printf("%d\t", a[i][j]);
    }
    printf("\n");
  }
  printf("\n");
}
void bfs(int a[][n], int i, int s, int visited[], int *c, struct q *qu, int
comp[], int pr[], int *cycfg)
{
  if (k == 0 && i != s)
  {
    visited[i] = (*c) * n;
    k = *c;
  }
  else
  {
    *(c) += 1;
    visited[i] = *c;
  }
  insert(qu, i);
  while (isEmpty(qu) != 1)
  {
    i = qu->arr[qu->f];
    for (int j = 0; j < n; j++)
    {
      count++;
      if (visited[j] == 0 && a[i][j] == 1)
      {
```

```
            *(c) += 1;
            visited[j] = *c;
            insert(qu, j);
            pr[j] = i;
        }
        else
        {
            if (a[i][j] == 1 && pr[i] != j)
                *(cycfg) = 1;
        }
    }
    printf("%c\t", i + 65);
    comp[k] = i;
    k += 1;
    del(qu);
  }
}
void bfs_help(int a[][n], int s, int visited[], int *c, struct q *qu, int
comp[], int pr[], int *cycfg)
{
  for (int i = 0; i < n; i++)
  {
    comp[i] = 0;
    visited[i] = 0;
    pr[i] = 0;
  }
  for (int i = s; (i % (n + 1)) < n; i++)
  {
    i = i % (n + 1);
    if (visited[i] == 0)
    {
      ct += 1;
      k = 0;
      bfs(a, i, s, visited, c, qu, comp, pr, cycfg);
    }
```

```
    }
}
void printComponent(int comp[], int visited[], int n)
{
    int i = 0, p = 1;
    while (i < n)
    {
        printf("Component %d is\n", p);
        p++;
        do
        {
            printf("%c\t", comp[i] + 65);
            i++;
        } while (visited[i] <= n && i < n);
        printf("\n");
    }
}
void main()
{
    FILE *f = fopen("bfs_count.txt", "w");
    int c = 0, s, cycfg = 0;
    char m;
    n = 4;
    struct q *qu = malloc(sizeof(struct q));
    while (n <= 6)
    {
        count = 0, c = 0, ct = 0, k = 0, cycfg = 0;
        init(qu, n);
        int a[n][n], b[n][n];
        int visited[n];
        int comp[n];
        int pr[n];

        // best case
        genData(a, n, 1);
```

```
printf("enter start node\n");
scanf(" %c", &m);
s = m - 65;
printf("Traversal\n");
bfs_help(a, s, visited, &c, qu, comp, pr, &cycfg);
printf("\n");
int i = 0;
if (ct > 1)
    printf("Disconnected\n");
else
    printf("Connected\n");
printf("Connected components are\n");
printComponent(comp, visited, n);
c = 0;
if (cycfg == 1)
    printf("Cyclic\n");
else
    printf("Acyclic\n");
printf("Count : %d", count);
fprintf(f, "%d\t%d\n", n, count);
free(qu->arr);

// worst case
init(qu, n);
count = 0, c = 0, ct = 0, k = 0, cycfg = 0;
genData(b, n, 0);
printf("enter start node\n");
scanf(" %c", &m);
s = m - 65;
printf("Traversal\n");
bfs_help(b, s, visited, &c, qu, comp, pr, &cycfg);
i = 0;
if (ct > 1)
    printf("Disconnected\n");
else
```

```
        printf("Connected\n");
    printf("Connected components are\n");
    printComponent(comp, visited, n);
    c = 0;
    if (cycfg == 1)
        printf("Cyclic\n");
    else
        printf("Acyclic\n");
    n += 1;

    free(qu->arr);
    printf("Count : %d", count);
    fprintf(f, "%d\t%d\n", n, count);
    printf("\n");
  }
}
```

9 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.

```
#include <stdio.h>
#include<stdlib.h>
int count;
int n,k=0;
int dfs(int a[][n],int visited[],int topoorder[],int i,int p[])
{
  if(visited[i]==0) visited[i]=1;
  p[i]=1;
  for(int j=0;j<n;j++)
  {
    count++;
```

```
        if(a[i][j]==1)
          {
           if(visited[j]==1 && p[j]==1) return 0;
           else
           {
               if(visited[j]==0)
               {
                   if(dfs(a,visited,topoorder,j,p)==0) return 0;
               }
           }
        }
    }
    p[i]=0;
    topoorder[k]=i;
    k+=1;
    return 1;
}
int topo(int a[][n],int visited[],int topoorder[],int p[])
{
    for(int i=0;i<n;i++)
    {
        visited[i]=0;
    }
    for(int i=0;i<n;i++)
      {
        if(visited[i]%(n+1)==0)
        {
         if(dfs(a,visited,topoorder,i,p)==0) return 0;
        }
    }
    return 1;
}
void main()
{
    count=0;
```

```
    FILE *f;
    f = fopen("dfstopo.txt", "a");
    printf("Enter no. of nodes\n");
    scanf("%d",&n);
    int a[n][n];
    int topoorder[n];
    printf("Enter graph\n");
    for(int i=0;i<n;i++)
    {
     for(int j=0;j<n;j++)
     {
      scanf("%d",&a[i][j]);
      }
    }
    int visited[n];
    int p[n];
    int r=topo(a,visited,topoorder,p);
    fprintf(f, "%d\t%d\n", n, count);
    if(r!=0)
    {
        for(int i=n-1;i>=0;i--) printf("C%d\t",topoorder[i]);
        printf("\n");
    }
    else printf("Cyclic graph\n");
}
```

10 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.

```
// #include <stdio.h>
// #include <stdlib.h>
// typedef struct LinkedList
// {
//     int val;
//     struct LinkedList *next;
// } LL;
// LL *init(int n)
// {
//     LL *list = (LL *)malloc(n * sizeof(struct LinkedList));
//     return list;
// }
// LL *createNode(int k)
// {
//     LL *node = (LL *)malloc(sizeof(struct LinkedList));
//     node->val = k;
//     node->next = NULL;
//     return node;
// }
// void insert(LL *list, int v, int k)
// {
//     LL *temp = (list + v);
//     while (temp->next != NULL)
//     {
//         temp = temp->next;
//     }
//     LL *node = createNode(k);
//     temp->next = node;
// }
// void printList(LL *list, int n)
// {
//     for (int i = 0; i < n; i++)
//     {
```

```
//        LL *temp = (list + i);
//        while (temp->next != NULL)
//        {
//            printf("%d-->", temp->val);
//            temp = temp->next;
//        }
//        printf("%d-->X", temp->val);
//        printf("\n");
//    }
// }
// LL *findSrc(LL *list, int n)
// {
//    LL *src, *temp;
//    int f = 0;
//    for (int i = 0; i < n; i++)
//    {
//        if ((list + i)->val != -1)
//            src = (list + i);
//        else
//            continue;
//        for (int j = 0; j < n; j++)
//        {
//            if (j == i)
//                continue;
//            if ((list + j)->val != -1)
//                temp = (list + j)->next;
//            else
//                continue;
//            while (temp != NULL)
//            {
//                if (temp->val == src->val)
//                {
//                    f = 1;
//                    break;
//                }
```

```
//                else
//                    temp = temp->next;
//            }
//            if (f == 1)
//                break;
//        }
//        if (f != 1)
//            return src;
//        else
//            f = 0;
//    }
//    return NULL;
// }
// void rmvSrc(LL *list, LL *src, int n)
// {
//    LL *temp;
//    for (int i = 0; i < n; i++)
//    {
//        if (i != src->val)
//            temp = (list + i);
//        else
//            continue;
//        while (temp->next != NULL)
//        {
//            if (temp->next->val == src->val)
//                temp->next = temp->next->next;
//            else
//                temp = temp->next;
//        }
//    }
// }
// void main()
// {
//    int n, k, ct = 0;
//    printf("Enter no of vertices\n");
```

```
//     scanf("%d", &n);
//     LL *list = init(n);
//     for (int i = 0; i < n; i++)
//     {
//         (list + i)->val = i;
//         (list + i)->next = NULL;
//     }
//     printf("Enter Adjacency Matrix\n");
//     for (int i = 0; i < n; i++)
//     {
//         for (int j = 0; j < n; j++)
//         {
//             scanf("%d", &k);
//             if (k == 1)
//             {
//                 insert(list, i, j);
//             }
//         }
//     }
//     printList(list, n);
//     LL *sr;
//     printf("Topological order\n");
//     do
//     {
//         ct += 1;
//         sr = findSrc(list, n);
//         if (sr == NULL)
//             break;
//         printf("%d\t", sr->val);
//         rmvSrc(list, sr, n);
//         sr->val = -1;
//     } while (sr != NULL);
//     if (ct != n + 1)
//         printf("\nCycle detected\n");
// }
```

```
#include <stdio.h>

int graph[40][40], n, visited[40]={0}, indegree[40]={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(){
	int i,j,count=0;

	createGraph();

	for(i=0;i<n;i++){
		for(j=0;j<n;j++){
			if (graph[j][i]) indegree[i]++;
		}
	}

	printf("Topologically Sorted Order:\n");
	while(count<n){
		for(i=0;i<n;i++){
			if (!visited[i] && !indegree[i]){
				printf("%d-->",i);
				visited[i]=1;
				for(j=0;j<n;j++){
					if (graph[i][j]){
						graph[i][j]=0;
```

```
                        indegree[j]--;
                    }
                }
                count++;
                break;
            }
        }
    }
    printf("\n");

}
```

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

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int count = 0;
void swap(int *a, int *b)
{
   int temp = *a;
   *a = *b;
   *b = temp;
}
void heapify(int a[], int n, int i)
{
   int largest = i;
   int left = 2 * i;
   int right = 2 * i + 1;

   count++;
```

```
    if (left <= n && a[left] > a[largest])
        largest = left;

    if (right <= n && a[right] > a[largest])
        largest = right;

    if (largest != i)
    {
        swap(a + i, a + largest);
        heapify(a, n, largest);
    }
}

void heapsort(int a[], int n)
{
    int i;
    for (i = n / 2; i >= 1; i--)
        heapify(a, n, i);
    for (i = n; i >1; i--)
    {
        swap(a + 1, a + i);
        heapify(a, i-1, 1);
    }
}

int main()
{



    int *a, i, n;
    srand(time(0));
    FILE *fpc = fopen("heapcount.txt", "w");
    for (n = 10; n <= 100; n += 10)
    {
        a = (int *)malloc((n + 1) * sizeof(int));
```

```c
        // Best case All same elements
        count = 0;
        for (i = 1; i <= n; i++)
            a[i] = n;
        heapsort(a, n);
        fprintf(fpc, "%d\t%d\t", n, count);

        // Average Case Random elements
        for (i = 1; i <= n; i++)
            a[i] = rand() % 100;
        count = 0;
        heapsort(a, n);
        fprintf(fpc, "%d\t", count);

        // Worst case Increasing order
        for (i = 1; i <= n; i++)
            a[i] = i;
        count = 0;
        heapsort(a, n);
        fprintf(fpc, "%d\n", count);
    }
    fclose(fpc);
    free(a);
    return 0;
}
```

```
set title 'Heap  Plot'
set xrange [10:100]
set yrange [10:1000]
set xlabel 'Input'
set ylabel 'Count'
set style data linespoints
plot 'heapcount.txt' u 1:2 w lp,'heapcount.txt' u 3 w lp,'heapcount.txt'
u 4 w lp
```

12 a) Implement Warshall’s Algorithm to find the transitive closure of a directed graph and perform its analysis giving minimum 5 graphs with different number of vertices and edges. (starting with 4 vertices).

```
#include <stdio.h>
#include <stdlib.h>
int n;
int count;
void Warshall(int M[n][n], int n)
{
    printf("\n Inside Warshall \n");
    int i,j,k;
    int R[n][n];
    for (i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            R[i][j]=M[i][j];
        }
    }
    printf("R0 :\n");
    for (i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            printf("%d ",R[i][j]);
        }
        printf("\n");
    }
    for(k=0;k<n;k++)
    {
        for(i=0;i<n;i++)
        {
            count++;
```

```
            if(R[i][k]==1)
            {
                for(j=0;j<n;j++)
                {
                    count++;
                    if(R[i][j]!=1)
                    {
                        R[i][j]=(R[i][k]&&R[k][j]);
                    }
                }
            }
        }
    }
    printf("Final Graph :\n");
    for (i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            printf("%d ",R[i][j]);
        }
        printf("\n");
    }

}
void checkdirected(int a[n][n])
{
    int flag = 0;
    for (int i = 1; i <= n - 1; i++)
        for (int j = i; j <= n; j++)
            if (a[i][j] == 1 && a[j][i] == 1)
            {
                flag = 1;
                break;
            }
    if (flag == 1)
```

```
        printf("Warshall Algorithm not Applicable because graph is
undirected\n");
    else
        Warshall(a,n);
}


void main()
{
    FILE*f=fopen("Warshalls_data.txt","a");
    count=0;
    int i,j;
    printf(" Welocme \n Enter the number of Vertices : ");
    scanf("%d",&n);
    int a[n][n];
    printf(" Enter the Adjacency Matrix : \n");
    for (i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    printf("The Adjacency Matrix is : \n");
    for (i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            printf("%d ",a[i][j]);
        }
        printf("\n");
    }
    checkdirected(a);
    fprintf(f,"%d\t%d",n,count);
    fclose(f);
```

```
}
```

b) Implement Floyd’s Algorithm to find All-pair shortest paths for a graph and perform its analysis giving minimum 5 graphs with different number of vertices and edges(starting with 4 vertices).

```
#include <stdio.h>
#include <stdlib.h>
int n;
int count;
int min(int x,int y)
{
  if(x<y) return x;
  else return y;
}

void Floyd(int M[n][n], int n)
{
    printf("\n Inside Warshall \n");
    int i,j,k,T;
    int D[n][n];
    for (i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            D[i][j]=M[i][j];
        }
    }
    printf("D0 :\n");
    for (i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
```

```
            printf("%d ",D[i][j]);
        }
        printf("\n");
    }
    for(k=0;k<n;k++)
    {
        for(i=0;i<n;i++)
        {
            T=D[i][k];
            for(j=0;j<n;j++)
            {
                count++;
                if(D[i][j]>T)
                {
                    D[i][j]=min(D[i][j],T+D[k][j]);
                }
            }

        }
    }
    printf("Final Graph :\n");
    for (i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            printf("%d ",D[i][j]);
        }
        printf("\n");
    }

}

void main()
{
    FILE*fptr =fopen("Floyd_count.txt","a");
```

```
    count=0;
    int i,j;
    printf(" Welocme \n Enter the number of Vertices : ");
    scanf("%d",&n);
    int a[n][n];
    printf(" Enter the Adjacency Matrix : \n");
    for (i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    printf("%d ",a[2][2]);
    printf("The Adjacency Matrix is : \n");
    for (i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            printf("%d ",a[i][j]);
        }
        printf("\n");
    }
    Floyd(a,n);
    fprintf(fptr,"%d\t%d",n,count);
    fclose(fptr);
}
```

13 a) Implement bottom up Dynamic Programming algorithm to solve Knapsack problem and perform its analysis with different instances (different number of items and Capacity, starting with 4 items)

```c
#include <stdio.h>
#include <stdlib.h>
int i, j, n, c, count;
int maximum(int a, int b)
{
    return (a > b) ? a : b;
}
void composition(int f[][c + 1], int w[])
{
    int num = 0;
    int subset[n];
    j = c;
    for (i = n; i >= 1; i--)
    {
        if (f[i][j] != f[i - 1][j])
        {
            subset[num++] = i;
            j -= w[i];
        }
    }
    printf("Composition is : ");
    for (i = 0; i < num; i++)
        printf("%d ", subset[i]);
    printf("\n");
}
void knapsack(int f[n + 1][c + 1], int w[], int v[])
{
    for (i = 0; i <= n; i++)
        f[i][0] = 0;
    for (j = 1; j <= c; j++)
        f[0][j] = 0;
    for (i = 1; i <= n; i++)
        for (j = 1; j <= c; j++)
        {
            count++;
```

```
            if (j - w[i] >= 0)
                f[i][j] = maximum(f[i - 1][j], v[i] + f[i - 1][j - w[i]]);
            else
                f[i][j] = f[i - 1][j];
        }
    printf("The Optimal Solution is %d\n", f[n][c]);
}

int main()
{
    printf("Enter the number of items : ");
    scanf("%d", &n);
    printf("Enter the Capacity : ");
    scanf("%d", &c);
    int w[n], v[n], f[n + 1][c + 1];
    printf("Enter the Items' Weights and Value\n");
    printf("Item\tWeight\tValue\n");
    for (i = 1; i <= n; i++)
    {
        printf("%d\t", i);
        scanf("%d\t%d", &w[i], &v[i]);
    }
    knapsack(f, w, v);
    composition(f, w);
    printf("Operation Count : %d\n", count);
    return 0;
}
```