SANGHVI INNOVATIVE ACADEMY

Institute of Technology & Management

INDEX

## Introduction

1 . What is Algorithm?

* A clearly specified set of simple instructions to b followed to solve a problem
  - Takes a set of values, as input and
  - produces a value, or set of values, as output

*may be specified

- In English as a computer program
- As a pseudocode

2. Data structures

   * Methods of organizing data

3. Program = algorithms + data structures

4 . Why need algorithm analysis?

   a. writing a working program is not good enough

   b. The program may be inefficient!

   c. If the program is run on a large data set, then the running time becomes an issue

**Algorithm Analysis**

* We only analyze correct algorithms / programs
* An algorithm is correct
  - If, for every input instance, it halts with the correct output
* Incorrect algorithms
  - Might not halt at all on some input instances
  - Might halt with other than the desired answer
* Analyzing an algorithm
  - Predicting the resources that the algorithm requires
  - Resources include
    => Memory
    => Communication bandwidth
    => Computational time (usually most important)
* Factors affecting the running time
  - Computer
  - Compiler
  - algorithm used
  - input to the algorithm
    => The content of the input affects the running time
    => Typically, the input size (number of items in the input) is the main consideration

- E.g. sorting problem => the number of items to be sorted
- E.g. multiply two matrices together => the total number of elements in the two matrices

* Machine model assumed

- ➢ Instructions are executed one after another, with no concurrent operations => Not parallel computers

Example

Calculate

$$\sum_{i=1}^{N} i^3$$

Algorithm to solve this example:

```
int sum(int n)
    {
          int psum
          psum=0;                    // Line 1
          for(int i=0;i<=n;i++)      // Line 2
           psum=psum+i*i*i;          // Line 3
          return psum;               // Line 4        }
```

This is how we calculate time complexity

* Lines 1 and 4 count for one unit each

* Line 3: executed N times, each time four units

* Line 2: (1 for initialization, N+1 for all the tests, N for all the increments) total 2N + 2

* total cost: 6N + 4 => O(N)

Time complexity can be classified as:

* Worst case running time of an algorithm

- ➢ The longest running time for any input of size n
- ➢ An upper bound on the running time for any input guarantee that the algorithm will never take longer
- ➢ Example: Sort a set of numbers in increasing order; and the data is in decreasing order
- ➢ The worst case can occur fairly often
  - E.g. in searching a database for a particular piece of information

* Best case running time

- ➢ sort a set of numbers in increasing order; and the data is already in increasing order

* Average case running time

- ➢ May be difficult to define what "average" means, usually it is taken as average of best and worst values.

## PROGRAM 1:-

**AIM : Write a program to implement exponential function**

**ALGORITHM** :

```
Algorithm Power ( x, k )
{
    pow := 1 ;
    m := n ;
    z := x ;
    while ( m>0 )
        {
         while ( ( m mod z ) = 0 ) do
           {
                $m= \lfloor m/2 \rfloor$ ;
                $z = Z^2$ ;
           }
         m = m-1 ;
         pow := pow * z;
        }
        return pow ;
}
```

**PROGRAM :**

```
#include<stdio.h>
#include<conio.h>
int exponent(int,int);
void main()
{
int x,n;
clrscr();
printf("\n\n\t\tenter x= ");
scanf("%d",&x);
printf("\n\n\t\tenter n= ");
scanf("%d",&n);
printf("\n\n\t\texponent of %d to the power %d is %d",x,n,exponent(x,n));
getch();
}
int exponent(int x,int n)
{
int m,pow,z;
m=n;
pow=1;
z=x;
while(m>0)
{
while(m%2==0)
{
m=m/2;
z=z*z;
}
  m=m-1;
pow=pow*z;
 }
return pow;
}
```

**OUTPUT :**

```
        enter x= 2
        enter n= 5
        exponent of 2 to the power 5 is 32
```

**CONCLUSION :** The time-complexity is O(log n).

## PROGRAM : 2

**AIM : Write a program to implement binary search using recursion**

**ALGORITHM :**

```
Algorithm bsearch(a[],l,h,x)
{
if(l= =h) then
{
if(x= = a[l]) then return l;
else return -1;
}
else
{
mid : = (l+h)/2;
if(x= = a[mid]) then return mid;
else
{
if(x<a[mid]) then
return bsearch(a,l,mid-1,x);
else
return bsearch(a,mid+1,h,x);
}
}
}
```

**PROGRAM :**

```
#include<stdio.h>
#include<conio.h>
int a[10];
int bsearch(int[], int,int,int);
void main()
{
int i,l,h,x,y;
```

```
clrscr();
printf("\n\n\t\tenter l: ");
scanf("%d",&l);
printf("\n\n\t\tenter h: ");
scanf("%d",&h);
printf("\n\n\t\tenter elements : ");
for(i=l;i<=h;i++)
{
scanf("%d",&a[i]);
}
printf("\n\n\t\tenter element to be search: ");
scanf("%d",&x);
y=bsearch(a,l,h,x);
if(y==-1)
printf("\n\n\t\telement not found");
else
printf("\n\n\t\telement found at: %d ",y);
getch();
}
int bsearch(int a[],int l,int h,int x)
{
int mid;
if(l==h)
{
if(x==a[l])
return l;
else return -1;
}
else
{
mid=(l+h)/2;
if(x==a[mid])
return mid;
else if(x<a[mid])
bsearch(a,l,mid-1,x);
else
bsearch(a,mid+1,h,x);
}
}
```

**OUTPUT** :

```
enter l: 1
enter h: 10
enter elements : 1 3 5 7 9 11 13 15 17 19
enter element to be search: 15
element found at: 8
```

**CONCLUSION** : The complexity for the binary search using recursion is O(log n)

## PROGRAM :3

**AIM : Write a program to implement binary search using iteration**

**ALGORITHM** :

```
Algorithm Bsearch (a, l,h,x)
    {
        while (l≤h) do
          {
             mid := └(l+h)/2┘ ;
             if ( x= a[mid] ) then
                return mid ;
             else if ( x< a[mid] ) then
                h : = mid – 1;
                   else
                        l := mid +1
          }
             return 0 ;
          }
```

**PROGRAM :**

```
#include<stdio.h>
#include<conio.h>
int a[10];
int bsearch(int[], int,int,int);
void main()
{
int i,l,h,x,y;
clrscr();
printf("\n\n\t\tenter l: ");
scanf("%d",&l);
printf("\n\n\t\tenter h: ");
scanf("%d",&h);
printf("\n\n\t\tenter elements : ");
for(i=l;i<=h;i++)
{
scanf("%d",&a[i]);
}
printf("\n\n\t\tenter element to be search: ");
scanf("%d",&x);
y=bsearch(a,l,h,x);
if(y!=-1)
printf("\n\n\t\telement found at: %d ",y);
else
printf("\n\n\t\telement not found");
getch();
}
int bsearch(int a[],int l,int h,int x)
{
int mid;
while(l<=h)
{
mid=(l+h)/2;
if(x==a[mid])
return mid;
else if(x<a[mid])
h=mid-1;
else
l=mid+1;
}
return -1;
}
```

**OUTPUT :**

```
        enter l: 1
        enter h: 10
```

```
enter elements : 2 4 6 8 10 12 14 16 18 20
enter element to be search: 8
element found at: 4
```

**CONCLUSION :** The time-complexity is O(log n).

## PROGRAM :4

**AIM** : **Write a program to implement tower of Hanoi**

**ALGORITHM** :

```
Algorithm TowersofHanoi( n, x, y, z )
    \\ Move the top n disk from tower x to tower y
   {
        if ( n ≥ 1) then
        {
         TowerofHanoi( n-1 , x , z , y );
          write (“move top disk from tower”,x,”to top of tower”,y );
                 TowerofHanoi( n-1 , z  ,y , x );
         }
     }
```

**PROGRAM** :

```
 #include<stdio.h>
#include<conio.h>
void towers(int,char,char,char);
void towers(int n,char frompeg,char topeg,char auxpeg)
        { /* If only 1 disk, make the move and return */
         if(n==1)
           { printf("\nMove disk 1 from peg %c to peg    %c",frompeg,topeg);
             return;
           }
         /* Move top n-1 disks from A to B, using C as auxiliary */
         towers(n-1,frompeg,auxpeg,topeg);
         /* Move remaining disks from A to C */
         printf("\nMove disk %d from peg %c to peg %c",n,frompeg,topeg);
         /* Move n-1 disks from B to C using A as auxiliary */
         towers(n-1,auxpeg,topeg,frompeg);
        }
void main()
```

```
{
  int n;
 clrscr();
  printf("Enter the number of disks : ");
  scanf("%d",&n);
  printf("The Tower of Hanoi involves the moves :\n\n");
  towers(n,'A','C','B');
  getch();
}
```

**OUTPUT** :

```
Enter the number of disks : 4
The Tower of Hanoi involves the moves :
Move disk 1 from peg A to peg B
Move disk 2 from peg A to peg C
Move disk 1 from peg B to peg C
Move disk 3 from peg A to peg B
Move disk 1 from peg C to peg A
Move disk 2 from peg C to peg B
Move disk 1 from peg A to peg B
Move disk 4 from peg A to peg C
Move disk 1 from peg B to peg C
Move disk 2 from peg B to peg A
Move disk 1 from peg C to peg A
Move disk 3 from peg B to peg C
Move disk 1 from peg A to peg B
Move disk 2 from peg A to peg C
Move disk 1 from peg B to peg C
```

**CONCLUSION** :

The time-complexity is $O(2^a)$.

**Program :5**

**AIM : Write a program for maximum and minimum Using divide and conquer method**

**ALGORITHM** :

```
Algorithm MaxMin (i,j,max,min)
   \\ a[1:n] is a global array . Parameter  i and j are i and j are integer,
   \\1≤i≤j≤n. The effect is to set max and min to the largest and
   \\smallest value in a[i:j] , respectively
   {
     If(i=j) then max := min := a[i] ;\\small(p)
      else if ( i = j-1 ) then \\ another case of small(p)
         {
            If ( a[i] = a[j] ) then
            {
              max := a[j] ;
              min  := a[i] ;
             }
             else
             {
               max := a[i] ;
               min  := a[j] ;
              }
          }
          else
          {
           \\ If P is not small divide P into subproblems
           \\ Find where to split the set.
             mid  :=  └(i+j )/2 ┘ ;
               \\ Solve the sub problems
                 MaxMin ( i,mid, max,min);
                MaxMin( mid+1,j,max1,min1);
                  \\ combine the solution
                   if (max < max1 ) then
```

```
                        max := max1;
                       if ( min > min1 ) then
                        min := min1;
                        }
        }
```

**PROGRAM** :

```
 #include<stdio.h>
#include<conio.h>
int a[20],l,h,max,min,max1,min1;
void maxmin(int,int,int&,int &);
void main()
{
int i;
clrscr();
printf(" \n\n\n\t enter l:");
scanf("%d",&l);
printf(" \n\n\n\t enter h:");
scanf("%d",&h);
printf(" \n\n\n\t enter element:");
 for(i=l;i<=h;i++)
 {
 scanf(" %d",&a[i]);
 }
maxmin(l,h,max,min);
printf("\n\n\t\tmax element is %d  &  min element is %d",max,min);
getch();
}
void maxmin(int l ,int h,int &max,int &min)
{
int mid;
if(l==h)
{
max=a[l];
min=a[l];
}
else if(l==h-1)
{
        if(a[l]<a[h])
        {
        max=a[h];
        min=a[l];
        }
        else
        {
        min=a[h];
        max=a[l];
        }
}
else
{
mid=(l+h)/2;
maxmin(l,mid,max,min);
maxmin(mid+1,h,max1,min1);
if(max<max1)
max=max1;
if(min>min1)
min=min1;
}
}
```

**INPUT:**

```
22 13 5 8 15 60 17 31 47
```

**<u>OUTPUT</u>:**

Maximum: 60

Minimum : 8

**<u>CONCLUSION:</u>**

Thus we saw using StraightMaxMin algorithm, we need (N–1) + (N–2) = 2N –3 comparisons. If we solve the same problem using divide and conquer technique number of comparisons are reduced to 3n/2 – 2. This saves 25% comparisons.

**PROGRAM :6**

**AIM : Write a program to implement Merge sort**

**ALGORITHM :**

```
 Algorithm Mergesort (l,h)
  {
     If (l<h) then
       {
        mid := (l+h)/2;
        Mergesort (l, mid);
        Mergesort (mid+1, h);
        Merge (l,mid,h) ;
       }
   }
   Algorithm Merge (l,mid , h)
    {
          i := l ;
          j :=mid+1;
          k := l;
      while ( i ≤ mid and j ≤ h)
          {
             If ( a[i] < a[j]) then
                c[k++] := a[i++];
              else
                 c[k++] := a[j++];
           }
        while (i ≤ mid )
            {
                c[k++] := a[i++];
             }
           while (j ≤ h)
              {
                c[k++] := a[j++];
                }
            for (i =l to h) do
                {
                  a[i] := c[i];
                }
        }
```

**PROGRAM :**

```
 #include<stdio.h >
#include<conio.h>
int a[20];
void mergesort(int,int);
void merge(int,int,int);
void main()
{
int l,h,i;
clrscr();
printf("\n\n\t\t enter l: ");
scanf("%d",&l);
printf("\n\n\t\t enter h: ");
scanf("%d",&h);
printf("\n\n\t\t  enter elements of array: ");
for( i=l;i<=h;i++)
{
scanf("%d",&a[i]);
}
mergesort(l,h);
printf("\n\n");
for(i=l;i<=h;i++)
{
printf("  %d",a[i]);
}
```

```
getch( );
}
void mergesort(int l,int h)
{
int mid;
if(l<h)
{
mid=(l+h)/2;
mergesort(l,mid);
mergesort(mid+1,h);
merge(l,mid,h);
}
}
void merge(int l,int mid,int h)
{
int i,j,k,b[20];
i=l;
k=l;
j=mid+1;
while(i<=mid && j<=h)
{
if(a[i]<a[j])
{
b[k++]=a[i++];
}
else
{
b[k++]=a[j++];
}
}
while(i<=mid)
{
```

```
b[k++]=a[i++];
}
while(j<=h)
{
b[k++]=a[j++];
}
for(i=l;i<=h;i++)
{
a[i]=b[i];
}
}
```

**INPUT**: 13 5 8 15 60 17 31 47

**OUTPUT:**

Sorted numbers are: 8 5 13 15 17 31 47 60

**CONCLUSION:**

Performance Analysis of Merge Sort

Let T(n) be the time taken by this algorithm to sort an array of n elements dividing A into sub arrays A1 and A2 takes linear time. It is easy to see that the Merge (A1, A2, A) also takes the linear time. Consequently,

$T(n) = T(n/2) + T(n/2) + \theta(n)$

for simplicity

$T(n) = 2T(n/2) + \theta(n)$

The total running time of Merge sort algorithm is O(n lg n), which is asymptotically optimal like Heap sort, Merge sort has a guaranteed n lg n running time. Merge sort required (n) extra space. Merge is not in place algorithm. The only known ways to merge in place (without any extra space) are too complex to be reduced to practical program.

## **PROGRAM :7**

**AIM : write a program to implement Quicksort**

**ALGORITHM:**

```
Algorithm Quicksort (l,h)
{
    If ( l<h )
    {
     j := partition (a,l,h);
     Quicksort ( l,j-1 );
     Quicksort ( j+1 ,h);
    }
}
Algorithm partition (a,l,h)
 {
   key := a[l] ;
   i := l;
   j := h;
   while (i ≤ j)
     {
       while ( a[i] < key )
            i:= i+1;
         while ( a[j] > key )
            j:= j-1;
        if (i<j) then
          {
              swap (a,i,j);
          }
         else
           {
                swap (a,key ,j) ;
           }
      }
```

```
            return j ;
    }
```

**PROGRAM :**

```
#include<stdio.h>
#include<conio.h>
int a[20],i,j;
void quicksort(int,int);
int partition(int[],int,int);
void swap(int[],int *,int *);
void main()
{
int l,h;
clrscr();
printf("\n\t enter l:");
scanf("%d",&l);
printf("\n\t enter h:");
scanf("%d",&h);
printf("\n\t Enter elements of array:");
for(i=l;i<=h;i++)
{
scanf("%d",&a[i]);
}
printf("\n\t the arranged order is:\n");
quicksort(l,h);
for(i=l;i<=h;i++)
{
printf("\n\n%d",a[i]);
}
getch();
}
void quicksort(int l,int h)
```

```
{
if(l<h)
{
j=partition(a,l,h);
quicksort(l,j-1);
quicksort(j+1,h);
}
}
int partition(int a[],int l,int h)
{
int key,t;
key=a[l];
i=l;
j=h;
while(i<=j)
{
while(a[i]<=key)
{
i++;
}
while(a[j]>key)
{
j--;
}
if(i<j)
swap(a,&i,&j);
}
swap(a,&l,&j);
return j;
}
void swap(int a[],int *i,int *j)
{
```

```
int temp;
temp=a[*i];
a[*i]=a[*j];
a[*j]=temp;
}
```

**INPUT:** 13 5 8 15 60 17 31 47

**OUTPUT**:

```
Sorted numbers are: 8 5 13 15 17 31 47 60 27
```

**CONCLUSION:**

Performance Analysis of Quick Sort

The running time of quick sort depends on whether partition is balanced or unbalanced, which in turn depends on which elements of an array to be sorted are used for partitioning.

A very good partition splits an array up into two equal sized arrays. A bad partition, on other hand, splits an array up into two arrays of very different sizes. The worst partition puts only one element in one array and all other elements in the other array. If the partitioning is balanced, the Quick sort runs asymptotically as fast as merge sort. On the other hand, if partitioning is unbalanced, the Quick sort runs asymptotically as slow as insertion sort.

**Best Case**

The best thing that could happen in Quick sort would be that each partitioning stage divides the array exactly in half. In other words, the best to be a median of the keys in A[p . . r] every time procedure 'Partition' is called. The procedure 'Partition' always split

the array to be sorted into two equal sized arrays.

If the algorithm 'Partition' produces two regions of size n/2. The recurrence relation is

then

$T(n) = T(n/2) + T(n/2) + (n) = 2T(n/2) + (n)$

OR

$T(n) = (n \lg n)$

## PROGRAM :8

**AIM : Write a program to implement knapsack problem using greedy method**

**ALGORITHM :**

```
Algorithm GreedyKnapsack(m,n)
   \\ p[1:n] and w[1:n] contain the profit and weights respectively of the objects
 \\ordered such that p[i]/w[i] ≥ p[i+1]/w[i+1] . m is the knapsack size and x[1:n]
\\is the vector.
   {
        for i := 1 to n do
           x[i[ :=0.0;
         U := m;
         for i:= 1 to n do
             {
               if ( w[i] > U ) then break;
               x[i] := 1.0;
               U := U – w[i] ;
              }
            if ( i ≤ n ) then
              x[i] := U/w[i] ;
    }
```

**PROGRAM :**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,j,i,t,k;
float s=0.0;
float x[10],w[10],v[10],W,m,temp,amount,p[10],sol[10];
clrscr();
printf("\t\tFRACTIONAL KNAPSACK PROBLEM\n\n\n");
printf("ENTER NO. OF ITEMS\t:");
```

```
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("\nENTER WEIGHT OF ITEM A%d\t:",i);
scanf("%f",&w[i]);
printf("\nENTER PROFIT OF ITEM A%d\t:",i);
scanf("%f",&v[i]);
}
printf("\n\nENTER MAXIMUM CAPACITY OF KNAPSACK\t:");
scanf("%f",&m);
for(i=0;i<n;i++)
p[i]=v[i]/w[i];
for(i=0;i<n;i++)
for(j=0;j<n-1;j++)
{
if(p[j]<=p[j+1])
{
temp=p[j];
p[j]=p[j+1];
p[j+1]=temp;
temp=w[j];
w[j]=w[j+1];
w[j+1]=temp;
}
}
for(k=0;k<n;k++)
{
printf("\n\n\t%f   %f",p[k],w[k]);
}
for(i=0;i<n;i++)
{
x[i]=0.0;
}
W=m;
for(i=0;i<n;i++)
{
if(w[i]>W) break;
x[i]=1.0;
W=W-w[i];
}
if(i<=n-1)
x[i]=W/w[i];
printf("\n\nFINAL SOLUTION IS\n\n");
for(j=0;j<n;j++)
{
printf("x[%d]\t%f %f \n",j,x[j],w[j]*x[j]);
s=s+p[j]*x[j];
}
printf("\n\n\t\tmaximum profit is %f ",s);
getch();
}
```

**<u>OUTPUT</u> :**

```
            FRACTIONAL KNAPSACK PROBLEM
ENTER NO. OF ITEMS      :5
ENTER WEIGHT OF ITEM A0 :2
ENTER PROFIT OF ITEM A0 :3
ENTER WEIGHT OF ITEM A1 :5
ENTER PROFIT OF ITEM A1 :3
ENTER WEIGHT OF ITEM A2 :6
ENTER PROFIT OF ITEM A2 :85
ENTER WEIGHT OF ITEM A3 :4
ENTER PROFIT OF ITEM A3 :2
ENTER WEIGHT OF ITEM A4 :5
ENTER PROFIT OF ITEM A4 :2
ENTER MAXIMUM CAPACITY OF KNAPSACK      :20
      14.166667   6.000000
      1.500000   2.000000
      0.600000   5.000000
      0.500000   4.000000
      0.400000   5.000000
FINAL SOLUTION IS
x[0]    1.000000 6.000000
x[1]    1.000000 2.000000
x[2]    1.000000 5.000000
x[3]    1.000000 4.000000
x[4]    0.600000 3.000000
maximum profit is 17.006666
```

**CONCLUSION:**

**Performance Analysis**

If the items are already sorted into decreasing order of vi / wi, then the whileloop takes a time in O(n);

Therefore, the total time including the sort is in O(n log n).

If we keep the items in heap with largest vi/wi at the root. Then

· creating the heap takes O(n) time

· whileloop

now takes O(log n) time (since heap property must be restored after

the removal of root)

Although this data structure does not alter the worstcase,

it may be faster if only a small number of items are needed to fill the knapsack. One variant of the 01 knapsack problem is when order of items are sorted by increasing weight is the same as their order when sorted by decreasing value.

The optimal solution to this problem is to sort by the value of the item in decreasing order. Then pick up the most valuable item which also has a least weight. First, if its weight is less than the total weight that can be carried. Then deduct the total weight that can be carried by the weight of the item just pick. The second item to pick is the most valuable item among those remaining. Keep follow the same strategy until we cannot

carry more item (due to weight).

## PROGRAM: 9

**AIM : Write a program for finding all pair shortest path**

**ALGORITHM :**

```
Algorithm AllPSP( cost ,E ,n)
  {
            for i : =1 to n do
            for j : =1 to n do
                A[ i,j ] := cost[ i,j ];
           for k := 1 to n do
               for i := 1 to n do
                   for j : = 1 to n do
                       A[ i,j ] := min ( A[i,j] , A[i,k] +A[k,j]);
  }
```

**PROGRAM :**

```
 #include<stdio.h>
#include<conio.h>
#define infi 1000;
void main()
{
int n,i,j,k,a[10][10];
clrscr();
printf("\n\n\t\tenter number of node: ");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
printf("\n\tenter edge between %d  &  %d: ",i,j);
scanf("%d",&a[i][j]);
}
}
for(k=1;k<=n;k++)
{
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
if((a[i][j])<(a[i][k]+a[k][j]))
{
a[i][j]=a[i][j];
}
else
{
a[i][j]=a[i][k]+a[k][j];
}
}
}
}
getch();
clrscr();
printf("\n\n");
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
printf("\t%d",a[i][j]);
}
printf("\n\n\n");
}
getch();
}
```

**INPUT**:

```
        enter number of node: 4
    enter edge between 1  &  1: 12
    enter edge between 1  &  2: 10
    enter edge between 1  &  3: 4
    enter edge between 1  &  4: 2
    enter edge between 2  &  1: 6
    enter edge between 2  &  2: 55
    enter edge between 2  &  3: 1000
    enter edge between 2  &  4: 122
    enter edge between 3  &  1: 15
    enter edge between 3  &  2: 35
    enter edge between 3  &  3: 55
    enter edge between 3  &  4: 03
    enter edge between 4  &  1: 44
    enter edge between 4  &  2: 65
    enter edge between 4  &  3: 22
    enter edge between 4  &  4: 20
```

**OUTPUT** :

```
    12     10     4      2
    6      16     10     8
    15     25     19     3
    37     47     22     20
```

**CONCLUSION** :

· Time Complexity Analysis:

o The triple-for-loop that follows has a constanttime body, and thus takes O(n 3 ) time.

o Thus, the whole algorithm takes O(n 3 ) time.

## PROGRAM : 10

**AIM : Write a program to implement n- Queen problem**

**ALGORITM :**

```
Algorithm NQueen (k ,n )
    {
        for i := 1 to n do
          {
            if place ( k , i ) then
              {
                  x[k] := i;
                  if ( k = k ) then
                    write ( x[1:n]);
                  else
                    NQueen ( k+1,n)
              }
          }
    }
Algorithm place ( k,i)
    {
            for j := 1 to k-1 do
             {
             if ( (x[j] := i) or (abs(x[j]-1) := abs(j-k)) ) then
                return false ;
             else
                   return true;
             }
    }
```

**PROGRAM :**

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

```
#include<process.h>
int board[20];
int count;
void queen(int,int);
void printboard(int);
int place(int,int);
void main()
{
int n,i,j;
clrscr();
printf("\n\n\t\t enter number of queens: ");
scanf("%d",&n);
queen(1,n);
getch();
}
void printboard(int n)
{
int i,j;
printf("\n\nsolution %d:\n\n",++count);
        for(i=1;i<=n;i++)
        {
        printf("\t%d",i);
        }
        for(i=1;i<=n;i++)
        {
        printf("\n\n%d",i);
                for(j=1;j<=n;j++)
                {
                if(board[i]==j)
                printf("\tQ");
                else
                printf("\t-");
```

```
                        }
            }
printf("\n\n\t\tpress any key");
getch();
}
int place(int row,int column)
{
int i;
            for(i=1;i<=row-1;i++)
            {
             if(board[i]==column)
             return 0;
             else
             if(abs(board[i]-column)==abs(i-row))
             return 0;
            }
            return 1;
}
void queen(int row,int n)
{
int column;
for(column=1;column<=n;column++)
{
            if(place(row,column))
            {
                        board[row]=column;
                        if(row==n)
                        printboard(n);
                        else
                        queen(row+1,n);
            }
}
```

```
}
```

**OUTPUT :**

```
enter number of queens: 4
solution 1:
        1       2       3       4
1       -       Q       -       -
2       -       -       -       Q
3       Q       -       -       -
4       -       -       Q       -
                press any key
solution 2:
        1       2       3       4
1       -       -       Q       -
2       Q       -       -       -
3       -       -       -       Q
4       -       Q       -       -
                press any key
```

**CONCLUSION :**

All possible ways n queens can be placed on an n x n chessboard so that no two queens threaten each other. Each output consists of an array of integers col indexed from 1 to n, where col[i] is the column where the queen in the ith row is placed.

**PROGRAM:11**

**AIM: Write a program of Hamilton cycle**

**Algorithm :**

```
Algorithm Hamilton (k)
{
            repeat
                   {
                        Next value (k);
                        If(x[k]=0) then return;
                        If(k=n) then;
                                write x[1:n];
                        else
                               Hamilton(k+1);
                }until(false);
}

Algorithm Next value (k)
{
      repeat
            {
                      x[k] := (x[k]+1)mod(n+1);
                      if(x[k]=0) then return;
                      if(G[x[k-1],x[k]]≠0) then
                            {
                         For j:=1 to k-1 do
                               {
                                  If(x[j]:=x[k]) then break ;
                                  If(j:=k) then
                                    {
                                       If[(k<n) or [(k==n) and G(x[n],x[1]≠0)]] then
                                              Return;
                                                }
                        }
                    }
            }until(false);
    }
```

**PROGRAM:**

```
#include<stdio.h>
#include<conio.h>
#define max 25
int x[max];

void nextvertex(int G[max][max],int n,int k)
{
int j;

while(1)
{
        x[k]=(x[k]+1)%(n+1);
        if(x[k]==0)
        return;
        if(G[x[k-1]][x[k]]!=0)
        {
                for(j=1;j<=k-1;j++)
                {
                  if(x[j]==x[k])
                  break;

                }
            if(j==k)
            {
            if((k<n)||((k==n) && (G[x[n]][x[1]]!=0)))
            return;
```

```
                }
            }
}
}
void hcycle(int G[][max],int n,int k)
{
while(1)
{
            int i;
            nextvertex(G,n,k);
            if(x[k]==0)
            return;
            if(k==n)
            {

            printf("\n");
            for(i=1;i<=n;i++)
            printf("%d",x[i]);
            printf("%d",x[1]);
            }
            else
            hcycle(G,n,k+1);
}}
void main()
{
int i,j,v1,v2,Edges,n,G[max][max];
clrscr();
printf("\n\n\t\tenter no of vertices: ");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
            for(j=1;j<=n;j++)
            {
            G[i][j]=0;
            x[i]=0;
            }}
printf("\n\n\t\tenter no of edges: ");
scanf("%d",&Edges);
for(i=1;i<=Edges;i++)
{
printf("\n enter edge: ");
scanf("%d%d",&v1,&v2);
G[v1][v2]=1;
G[v2][v1]=1;
}
x[1]=1;
printf("\n hamiltonian cycle: \n");
hcycle(G,n,2);
getch()
}
```

**<u>OUTPUT</u>**

```
 enter no of vertices: 5
 enter no of edges: 6
 enter edge: 1 2
 enter edge: 1 3
 enter edge: 2 4
 enter edge: 2 5
 enter edge: 4 3
 enter edge: 4 5
 hamiltonian cycle:
125431
134521
```

**<u>CONCLUSION</u> :** The time-complexity is O($n^2$)

**PROGRAM:12**

**AIM: Program for traversing a graph through BFS and DFS**

**ALGORITHM:**

```
Algorithm BFS(v)
 // v is the initial vertex
// visited [i] := 1 if visited
// visited [i] =0 not visited
{
        u:=v;    visited [v] :=1    //q is a queue of unexplored vertices
        repeat
        {
                For all vertices w to u do
                {
                        If (visited[w]=0) then
                        {
                                Add w to q;  //w is unexplored
                                Visited of [w] := 1;
                        }
                }
                If q is empty then return ;      // no vertex explored
                Delete the next element u from q
                                        // wecan get the first unexplored element
        }until(false);
}
```

```
Algorithm DFS(v)
{
        Visited[v] := 1;
        For each vertex w adjacent to v do
        {
                If (visited[w]=0) then
```

```
                    DFS(w);
          }
}
```

**PROGRAM:**

```
#include<stdio.h>
#include<conio.h>
#define MAX 20

typedef enum boolean{false,true} bool;
int adj[MAX][MAX];
bool visited[MAX];
int n; /* Denotes number of nodes in the graph */
main()
{
int i,v,choice;

create_graph();
while(1)
{
printf("\n");
printf("1. Adjacency matrix\n");
printf("2. Depth First Search through recursion\n");
printf("3. Breadth First Search\n");
printf("4. Adjacent vertices\n");
printf("5. Exit\n");
printf("Enter your choice : ");
scanf("%d",&choice);

switch(choice)
{
case 1:
```

```
printf("Adjacency Matrix\n");
display();
break;
case 2:
printf("Enter starting node for Depth First Search : ");
scanf("%d",&v);
for(i=1;i<=n;i++)
visited[i]=false;
dfs_rec(v);
break;
case 3:
printf("Enter starting node for Breadth First Search : ");
scanf("%d", &v);
for(i=1;i<=n;i++)
visited[i]=false;
bfs(v);
break;
case 4:
printf("Enter node to find adjacent vertices : ");
scanf("%d", &v);
printf("Adjacent Vertices are : ");
adj_nodes(v);
break;
case 5:
exit(1);
default:
printf("Wrong choice\n");
break;
}/*End of switch*/
}/*End of while*/
}/*End of main()*/
```

```
create_graph()
{
int i,max_edges,origin,destin;
printf("Enter number of nodes : ");
scanf("%d",&n);
max_edges=n*(n-1);
for(i=1;i<=max_edges;i++)
{
printf("Enter edge %d( 0 0 to quit ) : ",i);
scanf("%d %d",&origin,&destin);
if((origin==0) && (destin==0))
break;
if( origin > n || destin > n || origin<=0 || destin<=0)
{
printf("Invalid edge!\n");
i--;
}
else
{
adj[origin][destin]=1;
}
}/*End of for*/
}/*End of create_graph()*/

display()
{
int i,j;
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
printf("%4d",adj[i][j]);
printf("\n");
```

```
}
}/*End of display()*/

dfs_rec(int v)
{
int i;
visited[v]=true;
printf("%d ",v);
for(i=1;i<=n;i++)
if(adj[v][i]==1 && visited[i]==false)
dfs_rec(i);
}/*End of dfs_rec()*/

bfs(int v)
{
int i,front,rear;
int que[20];
front=rear= -1;
printf("%d ",v);
visited[v]=true;
rear++;
front++;
que[rear]=v;

while(front<=rear)
{
v=que[front]; /* delete from queue */
front++;
for(i=1;i<=n;i++)
{
/* Check for adjacent unvisited nodes */
if( adj[v][i]==1 && visited[i]==false)
{
printf("%d ",i);
visited[i]=true;
rear++;
que[rear]=i;
}
}
}/*End of while*/
}/*End of bfs()*/

adj_nodes(int v)
{
int i;
for(i=1;i<=n;i++)
if(adj[v][i]==1)
printf("%d ",i);
printf("\n");
}/*End of adj_nodes()*/
```

**OUTPUT:**

```
Enter number of nodes : 8
Enter edge 1( 0 0 to quit ) : 1 2
Enter edge 2( 0 0 to quit ) : 1 3
Enter edge 3( 0 0 to quit ) : 2 4
Enter edge 4( 0 0 to quit ) : 2 5
Enter edge 5( 0 0 to quit ) : 3 6
Enter edge 6( 0 0 to quit ) : 3 7
Enter edge 7( 0 0 to quit ) : 4 8
Enter edge 8( 0 0 to quit ) : 5 8
Enter edge 9( 0 0 to quit ) : 6 8
Enter edge 10( 0 0 to quit ) : 7 8
Enter edge 11( 0 0 to quit ) : 0 0

1. Adjacency matrix
2. Depth First Search through recursion
3. Breadth First Search
4. Adjacent vertices
5. Exit
Enter your choice : 2
Enter starting node for Depth First Search : 1
1 2 4 8 5 3 6 7
1. Adjacency matrix
2. Depth First Search through recursion
3. Breadth First Search
4. Adjacent vertices
5. Exit
Enter your choice : 3
Enter starting node for Breadth First Search : 1
1 2 3 4 5 6 7 8
1. Adjacency matrix
2. Depth First Search through recursion
3. Breadth First Search
4. Adjacent vertices
5. Exit
Enter your choice : 1
Adjacency Matrix
   0   1   1   0   0   0   0   0
   0   0   0   1   1   0   0   0
   0   0   0   0   0   1   1   0
   0   0   0   0   0   0   0   1
   0   0   0   0   0   0   0   1
   0   0   0   0   0   0   0   1
   0   0   0   0   0   0   0   1
```

```
  0  0  0  0  0  0  0  0

1. Adjacency matrix
2. Depth First Search through recursion
3. Breadth First Search
4. Adjacent vertices
5. Exit
Enter your choice : 4
Enter node to find adjacent vertices : 5
Adjacent Vertices are : 8

1. Adjacency matrix
2. Depth First Search through recursion
3. Breadth First Search
4. Adjacent vertices
5. Exit
```

**CONCLUSION :** The time-complexity of BFS is O(V+2*E)

The time-complexity of DFS is O(V+E)