Design & Analysis of Algorithms
Semestral Examination

Asmita Samanta
Roll No.- CrS1902
Phone Number : 9051705371
Email id : asmitasamanta6@gmail.com
Web Page Link : https://github.com/AsmitaSamanta

August 2, 2020

1. (a) **Algorithm to calculate Least Common Multiple of two positive integer :**
Suppose two input integers are a and b, each of size n-bits.

   (i) If both two integers are 0, then it is impossible to calculate LCM. Now if any one of these two integers is 0, then the LCM will be 0.

   (ii) Now if both a and b are non zero,then we will first calculate $m = a \cdot b$ (multiply the given integers).

   (iii) Then we will compute GCD g of these two integers using Euclidean Algorithm.

   (iv) As we know
   (multiplcation of two integers) = (GCD of them) $\cdot$ (LCM of them).
   So we will compute LCM = (m/g).

   (b) Time complexity of multiplication of two integers of size n, is $O(n^2)$. [Actually here we sum n size integer n-many times, so this is of $O(n^2)$.]
   By same argument, time complexity of division of two integer of size atmost 2n and n is of $O(2n^2) = O(n^2)$.[In fact, a and b both are of size n, so clearly $a < 2^n$ and $b < 2^n$, and hence
   m $= a \cdot b < 2^{2n}$.]
   Now here we actually use 2n many division in the Euclidean Algorithm, so time complexity of calculating GCD is $O(n^3)$. So total time comlexity is $(O(n^2)+O(n^3)+O(n^2))$ i.e. the total time complexity is $O(n^3)$.

   (c) Yes my algorithm will work properly always after implementing in C. Because, (multiplcation of two integers) = (GCD of them) $\cdot$ (LCM of them) is true for all non zero integers.

I have implemented the algorithm in C as follows :

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

int main()
{
        int a,b,g,r,m,l;
        printf("Enter two integers : ");
        scanf("%d%d",&a,&b);
        if(a==0 && b==0)
                printf("\nL.C.M. is not possible !!!\n");
        else if(a==0 || b==0)
                printf("\nL.C.M. is 0.\n");
        else
        {
                printf("\nL.C.M. of %d and %d is ",a,b);
                m=a*b;
                r=a%b;
                while(r!=0)
                {
                        a=b;
                        b=r;
                        r=a%b;
                }
                g=b;
                l=m/g;
                printf("%d.\n",l);
        }
}
```

C program of cmputing LCM

2. (a) **Algorithm of Calculating Standard Deviation :**
Here input is n many integers of 4 bytes.
   (i) First we will compute mean of these integers using the formula $mean(\mu) = 1/n \cdot (\Sigma_{i=1}^{n} x_i)$.
   (ii) Then we will calculalate varience of those data using the frmula $varience(\sigma^2) = 1/n \cdot (\Sigma_{i=1}^{n} (x_i - \mu)^2)$.
   (iii) After that we will compute Standard Deviation of these integers by the formula $S.D.(\sigma) = \sqrt{varience} = \sqrt{\sigma^2}$.

Following this Algorithm I have written a piece of C code to compute Standard Deviation of n integers as follows :

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

int main()
{
        int *a,i,n,s1=0;
        float m,sd,s2=0;
        printf("\nEnter the number of integers : ");
        scanf("%d",&n);
        a=(int *)malloc(n*(sizeof(int)));
        printf("\nEnter the integers one by one : ");
        for(i=0;i<n;i++)
        {
                scanf("%d",&a[i]);
                s1=s1+a[i];
        }
        m=(s1*1.0)/n;
        printf("\nMean = %f.",m);
        for(i=0;i<n;i++)
                s2=s2+pow((a[i]-m),2);
        printf("\nVarience = %f.",(s2/n));
        sd=sqrt((s2/n));
        printf("\nStandard Deviation of the given integers are %f.\n",sd);
}
```

C program of cmputing Standard Deviation

(b) The maximum error for my code is 2^{-6}.

(c)

3. (a) I have written a c code to implement Merge Sort as follows :

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

void merge(int *a,int l,int m,int u)
{
        int i,j,k,*R,*S;
        R=(int *)malloc((m-l+1)*(sizeof(int)));
        S=(int *)malloc((u-m)*(sizeof(int)));
        for(i=0;i<(m-l+1);i++)
                R[i]=a[i+l];
        for(j=0;j<(u-m);j++)
                S[j]=a[j+m+1];
        i=0;
        j=0;
        k=l;
        while(i<m-l+1 && j<u-m)
        {
                if(R[i]<=S[j])
                {
                        a[k]=R[i];
                        i++;
                        k++;
                }
                else
                {
                        a[k]=S[j];
                        j++;
                        k++;
                }
        }
       while(i<m-l+1)
       {
               a[k]=R[i];
               i++;
               k++;
       }
       while(j<u-m)
       {
               a[k]=S[j];
               j++;
               k++;
       }
}


void mergesort(int *a,int l,int u)
{
        int m,i;
        if(l<u)
        {
                m=l+(u-l)/2;
                mergesort(a,l,m);
                mergesort(a,m+1,u);
                merge(a,l,m,u);
        }
}

int main()
{
        int *a,i,n;
        printf("\nPlease enter the number of elements : ");
        scanf("%d",&n);
        a=(int *)malloc(n*(sizeof(int)));
        printf("\nNow enter the elements one by one :\n");
       for(i=0;i<n;i++)
              scanf("%d",&a[i]);
       mergesort(a,0,n-1);

       printf("\nThe sorted array is :\n");
       for(i=0;i<n;i++)
               printf("%d  ",a[i]);
       printf("\n\n");
}
```

(b) Here I have considered an integer array x[ ] of size 18, where actually $x[0] = x_{-1} = 1$ and $x[1] = x_0 = 27(my\ day\ of\ birth)$.
Here it is told to create 16 integers using the formula
$x_i = (x_{i-1}^2 + 1)\ mod\ 97 + x_{i-2}$.
So in my case I have created x[2], x[3],..., x[17] using this formula as $x[i] = (x[i-1]^2 + 1)\ mod\ 97 + x[i-2]$.

My C-program is as follows :

```c
#include<stdio.h>
#include<math.h>
int main()
{
        int x[18],k,i;
        x[0]=1;
        printf("\n\tEnter your Day of Birth please : ");
        scanf("%d",&x[1]);
        for(i=2;i<18;i++)
        {
                k=pow((x[i-1]+1),2);
                x[i]=(k%97)+x[i-2];
        }
        printf("\n\tThe created integer series for you is :\n\n\t");
        for(i=0;i<18;i++)
                printf("%d  ",x[i]);
        printf("\n\n");
}
```

After running this code with input '27', I have got the following output :

```
asmita@asmita-Vostro-15-3568:~/Desktop/Asmita/book/2nd Sem/SM/Program$ gedit birth.c &
[2] 9949
asmita@asmita-Vostro-15-3568:~/Desktop/Asmita/book/2nd Sem/SM/Program$ gcc birth.c -o birth -lm
[2]+  Done                    gedit birth.c
asmita@asmita-Vostro-15-3568:~/Desktop/Asmita/book/2nd Sem/SM/Program$ ./birth

        Enter your Day of Birth please : 27

        The created integer series for you is :

        1  27  9  30  97  31  151  49  226  71  269  124  277  196  286  212  356  300

asmita@asmita-Vostro-15-3568:~/Desktop/Asmita/book/2nd Sem/SM/Program$
```

Now I use $x_1, x_2, ..., x_{10}$ i.e. in my case x[2], x[3],..., x[11] as input in my merge sort program and I have got the following output :

```
asmita@asmita-Vostro-15-3568:~/Desktop/Asmita/book/2nd Sem/SM/Program$ ./mrgsrt

Please enter the number of elements : 10

Now enter the elements one by one :
9  30  97  31  151  49  226  71  269  124

The sorted array is :
9  30  31  49  71  97  124  151  226  269

asmita@asmita-Vostro-15-3568:~/Desktop/Asmita/book/2nd Sem/SM/Program$
```

The Computation Steps are as follows :

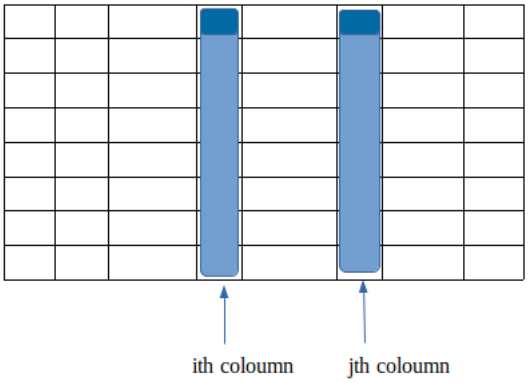
```
STEP 1 :        9  30  97  31  151  49  226  71  269  124
STEP 2 :        9  30  97  31  151
STEP 3 :        9  30  97
STEP 4 :        9  30
STEP 5 :        9
STEP 6 :        30
STEP 7 :        9  30
STEP 8 :        9  30
STEP 9 :        97
STEP 10 :       9  30  97
STEP 11 :       9  30  97
STEP 12 :       31  151
STEP 13 :       31
STEP 14 :       151
STEP 15 :       31  151
STEP 16 :       31  151
STEP 17 :       9  30  31  97  151
STEP 18 :       9  30  31  97  151
STEP 19 :       49  226  71  269  124
STEP 20 :       49  226  71
STEP 21 :       49  226
STEP 22 :       49
STEP 23 :       226
STEP 24 :       49  226
STEP 25 :       49  226
STEP 26 :       71
STEP 27 :       49  71  226
STEP 28 :       49  71  226
STEP 29 :       269  124
STEP 30 :       269
STEP 31 :       124
STEP 32 :       124  269
STEP 33 :       124  269
STEP 34 :       49  71  124  226  269
STEP 35 :       49  71  124  226  269
STEP 36 :       9  30  31  49  71  97  124  151  226  269
The sorted array is :
9  30  31  49  71  97  124  151  226  269
```

4. (a) **Algorithm :**

(i) First I will look at the 1st row of the matrix. I will create a n size 'node' array a[ ], where node is actually a data type, which contain 2 integer x, p, where a[i].x is value of the entry at M[1][a[i].p].

(ii) Then I will sort a[ ] in ascending order with respect to its x entry, i.e. we will check wheather $a[i].x < a[j].x$.

(iii) After full sorting of a[ ] we will check that which column of M goes where by p argument of a[ ]. Suppose a[i].p = j. Then we have to rewrite ith column, by jth column. To do this we will use an extra n-size integer array b[ ], I will copy ith column in b[ ],then I will copy jth column in ith column, and after that I will find the value of a[j].p and put that column in jth column and so on. Whenever I will get a[k].p =i, for some k, I will copy b[ ] in that kth column. I will do this process of copying untill all the columns are fully rearranged. I will actually start this process of copying for i=1.



copy jth column in ith column

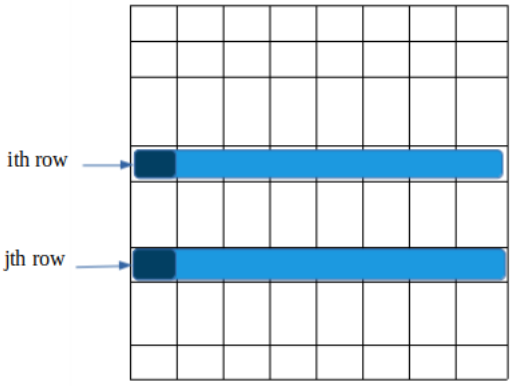
(iv) In this way I will get the 1st row in increasing order from left to right. Now we will call this permuted matrix as M_p.

(v) Now I will look at the 1st column of the permuted matrix M_p. I will put new values at the priveous array a[ ], where we will initially put a[i].x = M_p[i][1] and a[i].p = i.

(vi) Then we again sort a[ ] in decending order with respect to its

x entry, i.e. we will check wheather $a[i].x > a[j].x$.

(vii) After full sorting of a[ ] we will check that which row of M_p goes where by p argument of a[ ]. Suppose a[i].p = j. Then we have to rewrite ith row, by jth row. To do this we will use an extra n-size integer array b[ ], I will copy ith row in b[ ], then I will copy jth row in ith row, and after that I will find the value of a[j].p and put that row in jth row and so on. Whenever I will get a[k].p =i, for some k, I will copy b[ ] in that kth row. I will do this process of copying untill all the rows are fully rearranged. I will actually start this process of copying for i=1.



copy jth row in ith row

(viii) In this way I will get the 1st column of the ultimate permuted marix M' in decreasing order when we will look top to bottom and get one row, which was the 1st row of the intial permuted matrix M_p, in increasing order from left to right.

(b) The time complexity of my Algorithm is $O(n^2)$.
In fact, for both two sorting I need $O(n\ log\ n)$ many steps for each. After each sorting I am actually copying n^2 many elements. So total time complexity is $(2 \cdot O(n\ log\ n)) + O(n^2) = O(n^2)$.
No, I don't think this is optimal. Because here I choose 1st row of M, for sorting, but it may happen that if I choose any other row of M, then for sorting that row by permuting columns, I may need less steps. Similarly in M_p I choose 1st column for sorting, but it may happen that if I choose any other column of M_p, then for sorting that column by permuting rows, I may need less steps.

(c) For sorting of array a[ ], I will prefer to use Merge Sort Algorithm. Here we use an extra array a[ ] of size n but each a[i] containts 2 integers, So total space needed for a[ ] is 2n. For rearranging columns or rows we have used an extra n-size integer array, and for merge sort of a[ ], we need extra space wich is of $O(2n) = O(n)$. So the total extra space, which I need for my algorithm is $O(n)$.

5 (a) **Algorithm for All Pair Shortest Path :**
Let we have a graph G(V,E), where $|V|$=n. We also assume that it does not containt any negative weight cycle.

(i) Let $w(i, j)$ denotes the weight of the directed edge (i,j). At first we take a matrix W, where

$$\text{W[i][j]}= \begin{cases} 0 & \text{if } i = j \\ w(i,j) & \text{if } i \neq j \text{ and } (i,j) \in E \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E \end{cases}$$

(ii) We will introduce another matrix P. Let $p(i, j)$ = predicessor of j in our computed path from i to j. Then initially $p(i, j)$ = i, if there exist a directed edge from i to j. Now

$$\text{P[i][j]}= \begin{cases} NIL & \text{if } i = j \text{ or } \nexists \text{ path from } i \text{ to } j \\ p(i,j) & \text{if } i \neq j \text{ and } \exists \text{ path from } i \text{ to } j \end{cases}$$

(iii) Now we will consider a sequence of matrices $\text{L}^{(1)}$, $\text{L}^{(2)}$,..., $\text{L}^{(n-1)}$, where $\text{L}^{(m)}$[i][j] containts the total weight of maximum m-length shortest path from i to j. Initially $\text{L}^{(1)}$ is actually same with W.

(iv) To compute $\text{L}^{(m)}$ from $\text{L}^{(m-1)}$, we will actualy take $\text{L}^{(m)}\text{[i][j]} = min_{(k=1 \text{ to } n)} \{\text{L}^{(m-1)}\text{[i][k]} + \text{W[k][j]}\}$, and for that k, we will put P[i][j] = k, if j $\neq$ k, otherwise P[i][j] = P[i][j]. We denote the operation, by which we get $\text{L}^{(m)}$ using $\text{L}^{(m-1)}$ and W, as $\text{L}^{(m-1)} \cdot$ W.

(v) After finishing the coputation of $\text{L}^{(n-1)}$ we will get weight of the shortest length path from i to j with maximum (n-1)-many

edges is $L^{(n-1)}$[i][j]. Now to know the whole path from i to j, we will use the P matrix. We will do the path finding algorithm as follows :

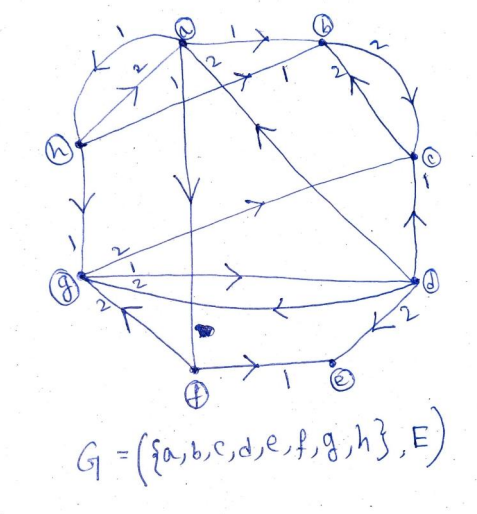
PrintPath(i,j)
1 **if** (i == j)
2 print i;
3 **else if** (P[i][j] == NIL)
4 print "There is no path from" i "to" j;
5 **else**
6 **PrintPath**(i,P[i][j])
7 print j;

In this way we can calculate All Pair Shortest Path of a given directed graph G(V,E).
But we can modify in this algorithm to get a first computation. Our main goal is not actually computing all the members of the sequence $L^{(1)}$, $L^{(2)}$,..., $L^{(n-1)}$, our main goal is to obtain $L^{(n-1)}$. Here actually we can notice that,
$L^{(1)} = W$,
$L^{(2)} = L^{(1)} \cdot W = W^2$,
$L^{(3)} = L^{(2)} \cdot W = W^3$,
.
.
.
$L^{(n-1)} = L^{(n-2)} \cdot W = W^{n-1}$
Again we know that here the operation "$\cdot$" is quite similar with matrix multiplication, in fact if we replace the operations 'mulplication' and 'sum' by the operations 'min' and 'sum' respectively in matrix multiplication, we will get our operation '$\cdot$'. So, as matrix multiplication, '$\cdot$' is also associative.
Again, since there is no negative weight cycle in our graph G, $L^{(m)} = L^{(n-1)}$ for all m $\geqslant$ (n-1).
So we will compute $L^{(1)}$, $L^{(2)}$, $L^{(4)}$..., $L^{(m)}$, $L^{(2m)}$, untill we get 2m > (n-1).
We will compute as follows :
$L^{(1)} = W$,
$L^{(2)} = L^{(1)} \cdot L^{(1)}$,
$L^{(4)} = L^{(2)} \cdot L^{(2)}$,
$L^{(8)} = L^{(4)} \cdot L^{(4)}$,

and so on ...
So we can finish the total computation in $\lceil log_2(n-1) \rceil$ steps in stead of (n-1) steps.
We also have to modify the computation of P matrix as follows :
In the time of conputing $L^{(2m)} = L^{(m)} \cdot L^{(m)}$, if we get
$L^{(2m)}[i][j] = L^{(m)}[i][k] + L^{(m)}[k][j]$, we will put $P[i][j] = P[k][j]$, if $j \neq k$, otherwise $P[i][j] = P[i][j]$.

(b) Here we need $\lceil log_2(n-1) \rceil$ many '$\cdot$' operations (which is quite similar with matrix multiplication, as I stated earlier) in my algorithm. In each '$\cdot$' operation, we need to compute n^2 many elements and we need $O(n)$ many operation to compute each of these elements. So in each '$\cdot$' operation, we need $O(n^3)$ many operations. So the total time complexity of my algorithm is
$O(n^3 \lceil log_2(n-1) \rceil) = O(n^3 \ log \ n)$.

(c) For example, I am taking the following graph :



Here the weight matrix W is as follows :

$$\mathrm{W} = \begin{bmatrix} 0 & 1 & \infty & \infty & \infty & 1 & \infty & 1 \\ \infty & 0 & 2 & \infty & \infty & \infty & \infty & \infty \\ \infty & 2 & 0 & \infty & \infty & \infty & \infty & \infty \\ 2 & \infty & 1 & 0 & 2 & \infty & 2 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 1 & 0 & 2 & \infty \\ \infty & \infty & 2 & 1 & \infty & \infty & 0 & \infty \\ 2 & 1 & \infty & \infty & \infty & \infty & 1 & 0 \end{bmatrix} = \mathrm{L}^{(1)}$$

And the path-parent matrix P is as follows :

$$\mathrm{P} = \begin{bmatrix} NIL & a & NIL & NIL & NIL & a & NIL & a \\ NIL & NIL & b & NIL & NIL & NIL & NIL & NIL \\ NIL & c & NIL & NIL & NIL & NIL & NIL & NIL \\ d & NIL & d & NIL & d & NIL & d & NIL \\ NIL & NIL & NIL & NIL & NIL & NIL & NIL & NIL \\ NIL & NIL & NIL & NIL & f & NIL & f & NIL \\ NIL & NIL & g & g & NIL & NIL & NIL & NIL \\ h & h & NIL & NIL & NIL & NIL & h & NIL \end{bmatrix}$$

Now we will compute $\mathrm{L}^{(2)}$, $\mathrm{L}^{(4)}$ and $\mathrm{L}^{(8)}$ and crresponding P matrices for each.

Step-1 :

$$\mathrm{L}^{(2)} = \mathrm{L}^{(1)} \cdot \mathrm{L}^{(1)} = \begin{bmatrix} 0 & 1 & 3 & \infty & 2 & 1 & 2 & 1 \\ \infty & 0 & 2 & \infty & \infty & \infty & \infty & \infty \\ \infty & 2 & 0 & \infty & \infty & \infty & \infty & \infty \\ 2 & 3 & 1 & 0 & 2 & 3 & 2 & 3 \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & 4 & 3 & 1 & 0 & 2 & \infty \\ 3 & 4 & 2 & 1 & 3 & \infty & 0 & \infty \\ 2 & 1 & 3 & 2 & \infty & 3 & 1 & 0 \end{bmatrix},$$

and the corresponding

$$\mathrm{P} = \begin{bmatrix} NIL & a & b & NIL & f & a & h & a \\ NIL & NIL & b & NIL & NIL & NIL & NIL & NIL \\ NIL & c & NIL & NIL & NIL & NIL & NIL & NIL \\ d & a & d & NIL & d & a & d & a \\ NIL & NIL & NIL & NIL & NIL & NIL & NIL & NIL \\ NIL & NIL & g & g & f & NIL & f & NIL \\ d & c & g & g & d & NIL & NIL & NIL \\ h & h & b & g & NIL & a & h & NIL \end{bmatrix}$$

Step-2 :

$$\mathrm{L}^{(4)} = \mathrm{L}^{(2)} \cdot \mathrm{L}^{(2)} = \begin{bmatrix} 0 & 1 & 3 & 3 & 2 & 1 & 2 & 1 \\ \infty & 0 & 2 & \infty & \infty & \infty & \infty & \infty \\ \infty & 2 & 0 & \infty & \infty & \infty & \infty & \infty \\ 2 & 3 & 1 & 0 & 2 & 3 & 2 & 3 \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ 5 & 6 & 4 & 3 & 1 & 0 & 2 & 6 \\ 3 & 4 & 2 & 1 & 3 & 4 & 0 & 4 \\ 2 & 1 & 3 & 2 & 4 & 3 & 1 & 0 \end{bmatrix},$$

and the corresponding

$$\mathrm{P} = \begin{bmatrix} NIL & a & b & g & f & a & h & a \\ NIL & NIL & b & NIL & NIL & NIL & NIL & NIL \\ NIL & c & NIL & NIL & NIL & NIL & NIL & NIL \\ d & a & d & NIL & d & a & d & a \\ NIL & NIL & NIL & NIL & NIL & NIL & NIL & NIL \\ d & c & g & g & f & NIL & f & a \\ d & a & g & g & d & a & NIL & a \\ h & h & b & g & f & a & h & NIL \end{bmatrix}$$

Step-3 :

$$L^{(8)} = L^{(4)} \cdot L^{(4)} = \begin{bmatrix} 0 & 1 & 3 & 3 & 2 & 1 & 2 & 1 \\ \infty & 0 & 2 & \infty & \infty & \infty & \infty & \infty \\ \infty & 2 & 0 & \infty & \infty & \infty & \infty & \infty \\ 2 & 3 & 1 & 0 & 2 & 3 & 2 & 3 \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ 5 & 6 & 4 & 3 & 1 & 0 & 2 & 6 \\ 3 & 4 & 2 & 1 & 3 & 4 & 0 & 4 \\ 2 & 1 & 3 & 2 & 4 & 3 & 1 & 0 \end{bmatrix},$$

and the corresponding

$$P = \begin{bmatrix} NIL & a & b & g & f & a & h & a \\ NIL & NIL & b & NIL & NIL & NIL & NIL & NIL \\ NIL & c & NIL & NIL & NIL & NIL & NIL & NIL \\ d & a & d & NIL & d & a & d & a \\ NIL & NIL & NIL & NIL & NIL & NIL & NIL & NIL \\ d & a & g & g & f & NIL & f & a \\ d & a & g & g & d & a & NIL & a \\ h & h & b & g & f & a & h & NIL \end{bmatrix}$$

Now actually using **PrintPath** algorithm and P matrix we can get All Pair Shortest path of the given graph G(V,E) and from $L^{(8)}$ we will get the weight of the path.

6. (a) Suppose we have two sequence. Then the longest common subsequence (LCS) problem is the problem of finding one of the longest subsequence common to both sequences. It is not same as the longest common substring problem because, like substrings, subsequences are not required to occupy in consecutive positions within the original sequences.
For example x and y are two integer sequences. take length of x is 8 and length of y is 5.
Let, x : 8 9 5 7 3 6 1 4 and y : 5 9 7 4 1. Then we have to find the longest common subsequence of x and y, along with the length.

(b) I have written the following C-program to compute LCS of given two sequences :

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int main()
{
        int i,j,k,l,m,n,**a;
        int *x,*y,*b;
        printf("\nEnter the length of the strings x and y :");
        scanf("%d%d",&m,&n);
        x=(int *)malloc(m*(sizeof(int)));
        y=(int *)malloc(n*(sizeof(int)));
        printf("\nEnter the string x :");
        for(i=0;i<m;i++)
                scanf("%d",&x[i]);
        printf("\nEnter the string y :");
        for(i=0;i<n;i++)
                scanf("%d",&y[i]);
        a=(int **)malloc((m+1)*(sizeof(int *)));
        for(i=0;i<=m;i++)
                a[i]=(int *)malloc((n+1)*(sizeof(int)));
        for(i=0;i<=m;i++)
                for(j=0;j<=n;j++)
                {
                        if(i==0 || j==0)
                                a[i][j]=0;
                        else
                        {
                                if(x[i-1]==y[j-1])
                                        a[i][j]=a[(i-1)][(j-1)]+1;
                                else
                                {
                                        if(a[i][(j-1)]<a[(i-1)][j])
                                                a[i][j]=a[(i-1)][j];
                                        else
                                                a[i][j]=a[i][(j-1)];
                                }
                        }
                }
        printf("\nThe computation matrix is :\n\n");
        for(i=0;i<=m;i++)
        {
                for(j=0;j<=n;j++)
                        printf("%d\t",a[i][j]);
                printf("\n\n");
        }
        printf("\nThe length of the longest subsequence is %d.\n",a[m][n]);
        i=m;
        j=n;
        l=a[m][n];
        b=(int *)malloc(l*(sizeof(int)));
        for(k=l;k>0;k--)
        {
                while(x[i-1]!=y[j-1])
                {
                        if(a[i-1][j]<a[i][j-1])
                                j--;
                        else
                                i--;
                }
                b[k-1]=x[i-1];
                i--;
                j--;
        }
        printf("One of the longest common subsequnce is : \t");
        for(i=0;i<l;i++)
                printf("%d  ",b[i]);
        printf("\n\n");
}
```

Now I give the input as of my example, i.e. length of x is 8 and length of y is 5 and x : 8 9 5 7 3 6 1 4 and y : 5 9 7 4 1. I got the output as follows :

```
asmita@asmita-Vostro-15-3568:~/Desktop/Asmita/book/2nd Sem/SM/Program$ gcc lcs.c -o lcs
asmita@asmita-Vostro-15-3568:~/Desktop/Asmita/book/2nd Sem/SM/Program$ ./lcs

Enter the length of the strings x and y :8 5

Enter the string x :8 9 5 7 3 6 1 4

Enter the string y :5 9 7 4 1

The computation matrix is :

0       0       0       0       0       0

0       0       0       0       0       0

0       0       1       1       1       1

0       1       1       1       1       1

0       1       1       2       2       2

0       1       1       2       2       2

0       1       1       2       2       2

0       1       1       2       2       3

0       1       1       2       3       3


The length of the longest subsequence is 3.
One of the longest common subsequnce is :       9  7  1

asmita@asmita-Vostro-15-3568:~/Desktop/Asmita/book/2nd Sem/SM/Program$
```

So, the length of Longest Common Subsequence is 3 and one of the Longest Common Subsequence is 9 7 1.

(c) Suppose let length of those two sequence is m and n respectively. To compute the matrix by comparing elements of two sequences we need $O(m \cdot n)$ time. After that to get back the common subsequence, in each step the sum i + j is decremented by at least 1 and stops as soon as one of i, j becomes zero; this is at least before i + j = 0, so we need at most (m + n) computation. So, the time complexity of finding Common Subsequence is $O(m+n)$. So, the total time complexity is $O(m \cdot n)$.

7. (a) Greedy algorithm is a simple algorithm which we use to solve optimization problems. As the word 'Greedy' suggests, this algorithm always makes a choice such that it would be the best solution at that moment. This means that it makes a local optimal choice

in each step such that this choice will become a global optimal solution.

(b) **Algorithm for Minimum Spanning tree :**
An undirected weighted connected graph G(V,E) is given. We take a set S in which we will put all the edges which will form a spanning tree.

(i) At first S = $\emptyset$.

(ii) For all $v \in$ V, do the operation Make-Set(v).

(iii) Sort the edges of E with respect to their weights in ascending order.

(iv) Now for each $e = (u, v)$ from the ordered set E (ordered with respect to weights in ascending) we will check wheather Find-Set(u) = Find-Set(v) or not

(v) If Find-Set(u) $\neq$ Find-Set(v), we will do S = S $\cup$ $\{(u, v)\}$ and Union(u, v).

(vi) In this way at last we will get a spanning tree for the connected graph G.

(d) We will need $O(|V|)$ time to do Make-Set operation for all vertices and $O(|E|\ log\ |E|)$ time to sort the edge set E. Now for each edge we are doing Find operation and if needed then Union operation, so it takes $O(|E|) \cdot O(log\ |V|) = O(|E|\ log\ |V|)$ time. Now clearly, $|E| < |V|^2$. So,$O(log\ |E|) = O(log\ |V|)$. So we get total time is $O(|V| + |E|\ log\ |V|)$. Again we also know that $|V| < |E|$, as the graph is connected. So the total time is $O(|E|\ log\ |V|)$.

9. (a) **3-SAT Problem :** Suppose we have a Boolean expression of the form $\Phi = (c_1 \wedge c_2 \wedge ... \wedge c_m)$, where for each $i \in \{1, 2, ..., m\}$, $c_i = (x_1 \vee x_2 \vee x_3)$ and $x_i = \overline{x_j}$ may happen for $i, j \in \{1, 2, 3\}$. Then 3-SAT problem is to find a truth assignments for all variables used in Φ such that Φ is true.

Vertex Cover Problem : Suppose we have a graph G(V,E), then vertex cover problem is to find wheather there is a vertex cover of G of size k (for a given k) or not.

(b) Suppose a graph G(V,E) has n vertices and I have told to find a k-size vertex cover (i.e. a vertex cover with k vertices). We

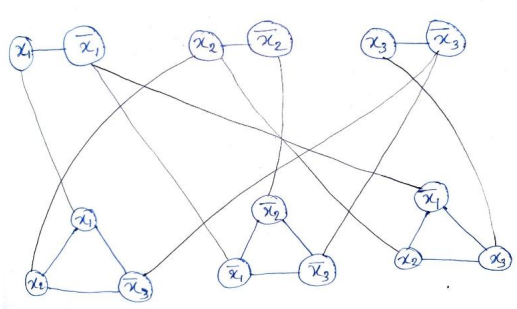
can choose any arbitrary k-subset of V and check wheather it is a vertex cover or not. For checking we need $O(|E|)$ time, and for choosing we have $\begin{pmatrix} n \\ k \end{pmatrix}$ many possibilities. So for choosing we need $O(n^k)$ time. So, clearly Vertex-Cover Problem is in NP.

Now we have to show that it is NP-hard. For this, we will show that if we can solve VertexCover Problem then it is very easy to solve 3-SAT Problem, and since 3-SAT is NP-hard, Vertex-Cover Problem is also NP-hard.

If possible let we can solve Vertex Cover Problem. Suppose we have given a 3-SAT problem Φ, then we will construct a graph G(V,E) from it as follows :
(i) Take a node for each x_i used in Φ and their complements, join all x_i with $\overline{x_i}$. We will call this portion as Variable Gadget.
(ii) For each clause in Φ we will draw a triangle whose nodes are the variables of that clause. We will call this portion as Clause Gadget.
(iii) We will join two nodes of variable Gadget and Clause Gadget iff they are same.

Here,
$|V| = 2\cdot(number\ of\ variables\ in\ \Phi)+3\cdot(number\ of\ clauses\ in\ \Phi)$.
we choose $k = (number\ of\ variables\ in\ \Phi)+2\cdot(number\ of\ clauses\ in\ \Phi)$.

Now we will show this graph construction by an example :
Let, $\Phi = (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3)$. Then the graph will be as follows :



Graph Construction

Now we will show that this graph G has a vertex cover of size k iff Φ is satisfiable.

Suppose we get a vertex cover of size k for this graph G. Now take the nodes of Variable Gadget, which are in the Vertex Set and assign them "True".
Our claim is that for this truth assignment Φ will be satisfiable.
If not then there exists a clause whose all 3 variables are assigned with "False", i.e. there is a triangle in Clause Gadget whose all three nodes are connected with some nodes in Variable Gadget, which are not present in the vertex Cover. Since we take only two vertices from each triangle of Clause Gadget, then one edge, joining Variable Gadget and Clause Gadget remains uncovered, which is a contradiction. So, if we get a vertex cover of G of size k, then Φ is satisfiable.

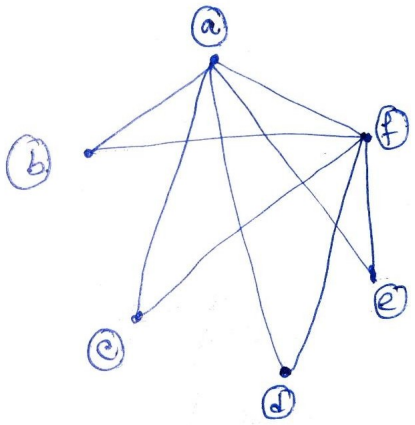
Now suppose Φ is satisfiable. Then we get a truth assignment for each variable and in Φ each clause has a variable assigned with "True".
Now we will construct a vertex cover of size k for G. To construct the vertex cover we will put all the nodes of Variable gadget whose variables are assigned with "True" in the vertex cover. Now in Clause gadget, we will put two nodes from each triangle in the vertex cover. We will take those two nodes as follows :
(i) If all three node variables are assigned with true, then we will take any two nodes.
(ii) If two node variables are "True" and one node variable is "False", then we take the node with False variable and any one node of true variables.
(iii) If exactly one node variable is assigned with "True", then take that two nodes with false variables.
So clearly by construction we can see that if Φ is satisfiable, then G has a vertex cover of size k.

So, 3-SAT Problem is a NP-hard problem and hence it is a NP-Complete Problem.

(c) Following this Algotithm we will actually get a vertex cover of

the graph, but not the particular vertex cover which is of size k. Because here we are just choosing an arbitrary vertex, without considering the degree of vertex.
I can explain this with the following example :

Suppose I have given this following graph for finding Vertex Cover using the algorithm $\mathcal{A}$.



This vertex actually has a vartex cover of size 2, which is {a, f} Now if we start the algorithm $\mathcal{A}$ with the vertex c, and after the first step we choose d, then we don't get any cover of size 2. In stead of that we will get a vertex cover like {c, d, b, a, e} (the choice of vertex cover is arbitrary so I just assume after choosing d, it will choose b, a and e respectively before termination).

So, we can say that the Algorithm $\mathcal{A}$ can find a vertex cover but it may not find the vertex cover of a particular size k.