Design and Analysis of Algorithms

Sarbajit Ghosh [CrS1911]

August 3, 2020

Email: gsarbajit@gmail.com
MNo: 8100242571
Website: https://github.com/ghosh-sarbajit/algo_end_sem_exam

1. (a) **Algorithm for LCM:**

Algorithm 1 LCM(n1,n2)

```
a ← n1
r ← n2
while r ≠ 0 do
    r ← a%b
    a ← b
    b ← r
end while
return (n1 * n2)/a
```

(b) **Complexity:** Given number is n-bit. So we can input $M = 2^n$ as highest number. And off course we have the following relation $n = \log_2 M$. Now let given two numbers as input are M_1 and M_2. Now if we consider the worst case analysis which occurs for Fibonacci sequence in that case the complexity might be $max\{M_1, M_2\}$ or $(O)(M)$. Now to obtain LCM form gcd we are multiplying and dividing two inserted numbers. Hence complexity for calculating multiplication and division is $(O)(M_1 * M_2)$ or for large numbers approximately $(O)(M^2)$.

(c) **C program for LCM:**

```c
#include <stdio.h>
int main ()
{
    int n1,n2,a,b,r,lcm;
    printf("%s","enter the integers: ");
    scanf("%d %d",&a,&b);
    n1=a;
    n2=b;
    r=b;
    while(r!=0)
    {
        r=a%b;
        a=b;
        b=r;
    }
```

```
    lcm = (n1*n2)/a;
    printf("lcm : %d", lcm);
    return 0;
}
```

Here we have used int type variable to store a number, a typical c compiler provides 4 bytes for storing an integer. Now in our algorithm we have to perform multiplication operation. Now for 2 n-bit integer their multiplication value will be upto 2-bit, hence there is a chance of integer overflow. So our program is able to compute LCM of both integers upto 2^16 if we take unsigned integers off course.

2. Not attempted.

3. (a) **Mergesort:**

```
#include <stdlib.h>
#include <stdio.h>
#define sentinel 10000
void merge(int *arr,int min,int mid,int max)
{
  int n1,n2;//size of sub array
  int i,j,k;//running index
  int *left,*right;//two sub array
  n1=mid-min+1;
  n2=max-mid;
  left=(int *)malloc(n1*sizeof(int));
  right=(int *)malloc(n2*sizeof(int));
  for(i=1;i<=n1;i++)
    *(left+i)=*(arr+min+i-1);
  for(i=1;i<=n2;i++)
    *(right+i)=*(arr+mid+i);
  *(left+n1+1)=sentinel;
  *(right+n2+1)=sentinel;
  i=1;
  j=1;
  for(k=min;k<=max;k++)
  {
    if(*(left+i)<=*(right+j))
    {
      *(arr+k)=*(left+i);
      i++;
    }
    else
    {
      *(arr+k)=*(right+j);
      j++;
    }
  }
}
void merge_sort(int *arr,int min,int high)
{
  int mid;
  if(min<high)
  {
    mid=((min+high)/2);
    merge_sort(arr,min,mid);
    merge_sort(arr,mid+1,high);
    merge(arr,min,mid,high);
  }
}
```

```c
int main()
{
  int *arr;
  int n;
  int i;
  printf("How many elements you want to give\n");
  scanf("%d",&n);
  arr=(int *)malloc(n*sizeof(int));
  printf("Enter array elements\n");
  for(i=0;i<n;i++)
  scanf("%d",arr+i);
  printf("Enterd array\n");
  for(i=0;i<n;i++)
  printf("%d\t", *(arr+i));
  printf("\n\n");
  merge_sort(arr,0,n-1);
  printf("Array after sorting\n");
  for(i=0;i<n;i++)
  printf("%d\t", *(arr+i));
  printf("\n\n");
  return 0;
}
```

(b) **Program to generate sequence:**

```c
#include <stdio.h>
#define BDAY 1
int main()
{
    int arr[20];
    arr[0]=-1;
    arr[1]=BDAY;
    for(int i=2;i<20;i++)
    {
        arr[i]=(arr[i-1]+1)*(arr[i-1]+1);
        arr[i]=arr[i]%(97+arr[i-2]);
    }
    for(int i=0;i<16;i++)
        printf("%d\t", arr[i]);
    printf("\n" );
    return 0;
}
```

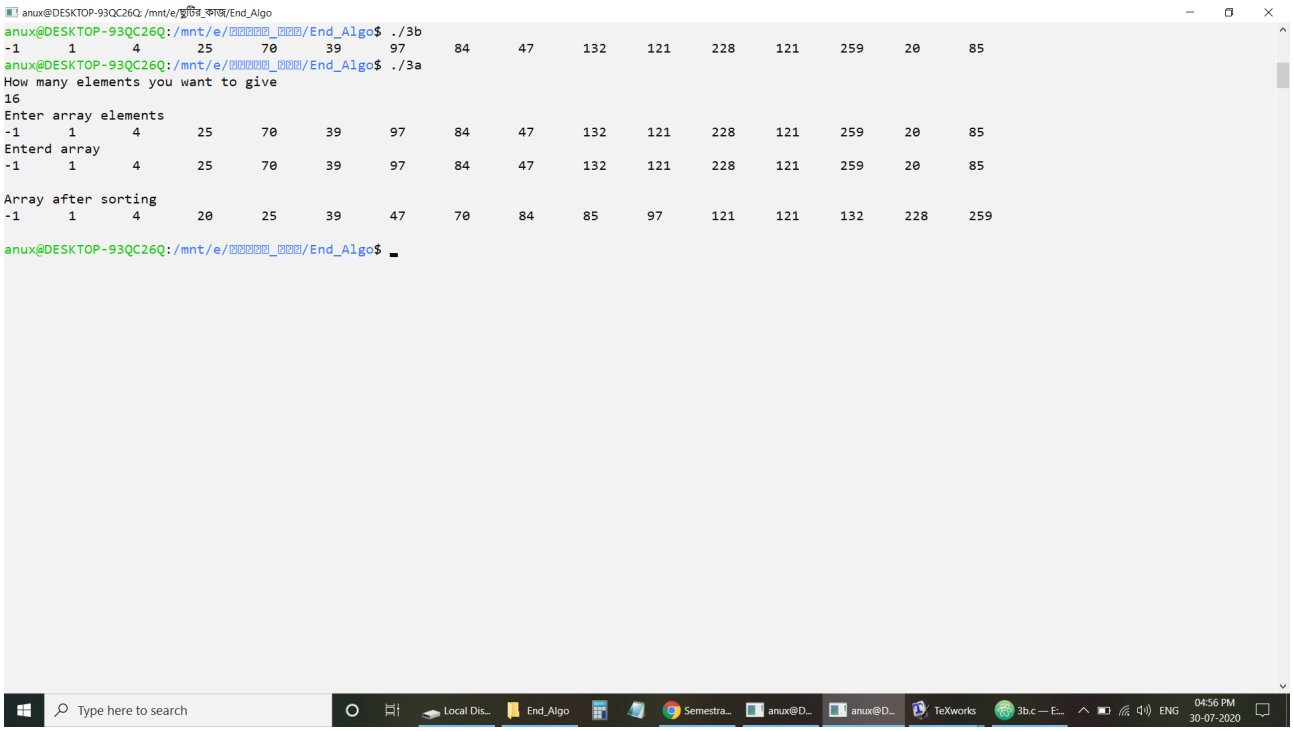

Figure 1: Sorting generated numbers, $x_{-1}, x_0, \ldots, x_{14}$

Explation with $x_1, x_2, \ldots, x_{10}$:

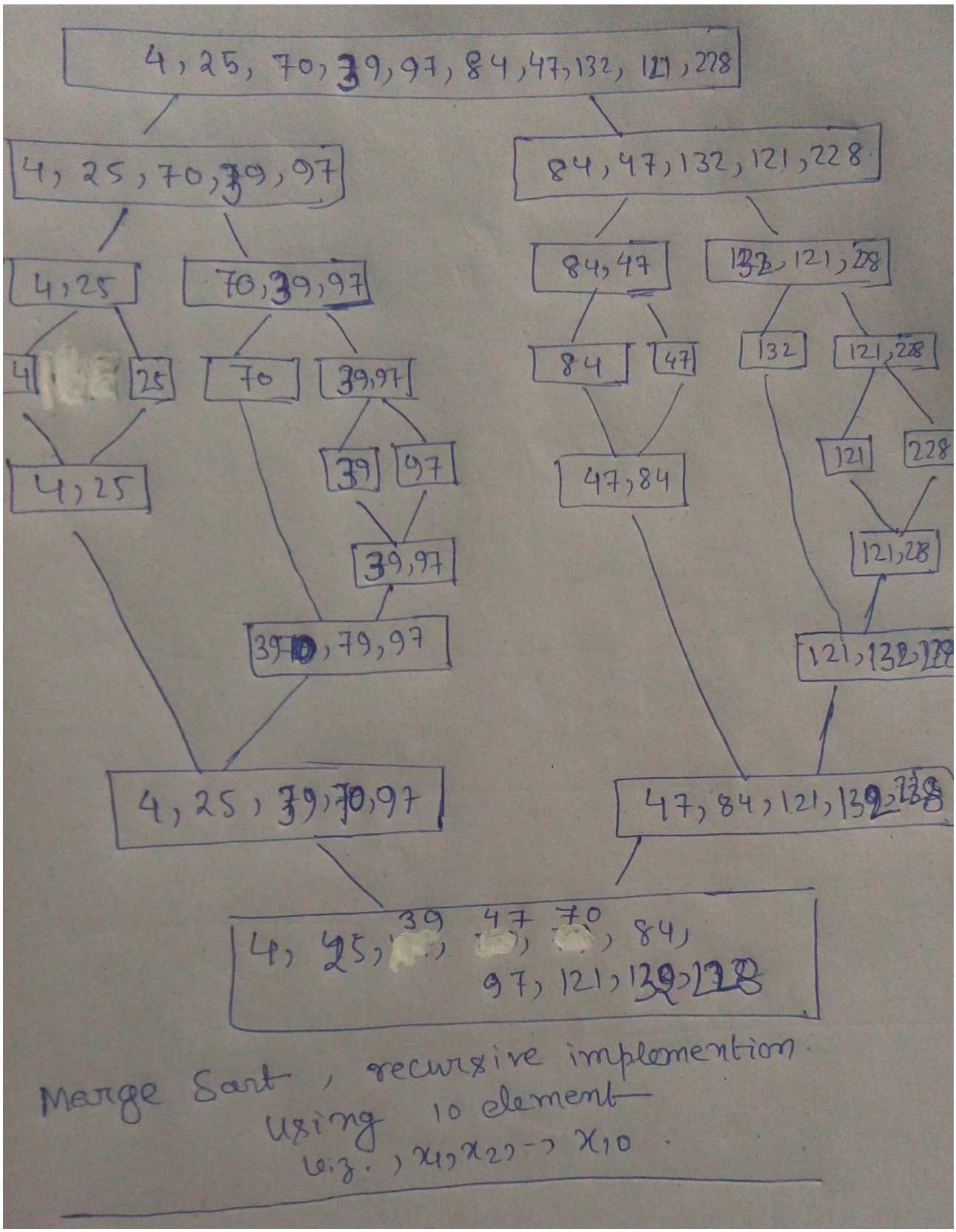

Figure 2: Sorting generated numbers, $x_1, x_2, \ldots, x_{10}$

4. (a) Here input is a $M_{n\times n}$ matrix.
Output is a $N_{n\times n}$ matrix, whose at least one row and and at least one column should be sorted. For row given condition is it should be sorted from left to right in increasing order, while for column it should be sorted in decreasing order from top

to bottom.

Algorithm 2 MatrixSort(M)

1: $r1, r2 \leftarrow Unif\{1:n\}$ ▷ generate two uniform random number from $1, 2, \ldots, n$
2: $Arr \leftarrow M[r1,]$ ▷ We take $r1^{th}$ row of M
3: $IArr \leftarrow [1:n]$ ▷ An array which contains 1:n
4: OurMergeSort(Arr,IArr) ▷ Internally calls merge sort, rearrange eles of IArr, required to permute cols of M
5: $M_1 \leftarrow Rearrange(M, IArr, 1)$ ▷ 0/1 row/col
6: $Arr \leftarrow M_1[, r2]$
7: $IArr \leftarrow [1:n]$
8: OurMergeSort(Arr,IArr)
9: $M_2 \leftarrow Rearranged(M_1, IArr, 0)$
10: **return** M_2

(b) **Time complexity:**

- For sorting we have $\mathcal{O}(n\log(n))$ comparison in line 4 and 8.
- By generating a permutation we are basically creating hash, and Rearrange function with help of that hash value record a particular row of column of its input matrix to its output matrix. So here for recording new elements the complexity is $\mathcal{O}(n^2)$
- Hence total time complexity is given by $\mathcal{O}(n^2 + n\log(n))$ which is $\mathcal{O}(n^2)$

Comment on optimality: Since all elements here are integer we can use radix sort to get more optimal result.

(c)

- Here we are using array to implement above algorithm.
- **Space complexity:** We are taking two matrix viz M_1, M_2, which will comprise of $\mathcal{O}(n^2)$ extra space. And $\mathcal{O}(n)$ amount of extra space for array. So total $\mathcal{O}(n^2)$.

5. (a) We will present **Floyed-Warshall** algorithm for all-pair shortest path.
Let us consider we have directed weighted graph (V, E, W). Where V, E set of vertex and edges respectively. $W = (w_{ij})$ is adjacency matrix. Let $d_{ij}^{(k)}$ br the weight of shortest path from vertex $i \to j$, for which all intermediate vertices are in $\{1, 2, \ldots, k\}$. We note that such a path is atleast of length one edge hence,

$$d_{ij}^{k} = \begin{cases} w_{ij}, & \text{if k=0} \\ min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\}, & \text{if k} \geq 1 \end{cases}$$

Since for any path, all intermediate vertices are in $\{1, 2, \ldots, k\}$ and $d_{ij}^{k} = \delta(i, j), \forall i, j \in V$ at end. We now present the algorithm.

Algorithm 3 Floyed-Warshall(W)

1: $n \leftarrow rows[W]$
2: $D^{(0)} \leftarrow W$
3: **for** $k \leftarrow 1(1)n$ **do**
4: **for** $i \leftarrow 1(1)n$ **do**
5: **for** $j \leftarrow 1(1)n$ **do**
6: $min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\}$
7: **end for**
8: **end for**
9: **end for**

(b) **Complexity:**In the above algorithm there are 3 loops, each takes $\mathcal{O}(1)$ time to execute hence its complexity is $\mathcal{O}(n^3)$

(c)

Here 8 is the number of vertices which is large, so we have write a c program to compute the cost and predecessor matrix. **Our program is as follows :**

```c
#include<stdio.h>
#include<stdlib.h>
#include <assert.h>
#define INF 100
//#define MIN(a,b) ((a<b)?a:b)

int node_num;
int **adj_mat, **wt_mat;
int **pi_mat, **D_mat;

int sum(int a,int b)
{
    if(a==INF || b==INF)
    {
        return INF;
    }
    else
    {
        return a+b;
    }
}

int MIN(int a,int b)
{
    if(a<=b)
    return a;
    else
    return b;
}

void pi_D_mat_calc(int iter)
{
    //printf("Hi\n" );
    int **mat,**mat1;
    int pred;
    mat=(int **)malloc(node_num*sizeof(int *));
    mat1=(int **)malloc(node_num*sizeof(int *));
  for(int i=0;i<node_num;i++)
  {
```

```
40      mat[i]= (int *)malloc(node_num*sizeof(int));
41      assert(mat[i] != NULL);
42          mat1[i]= (int *)malloc(node_num*sizeof(int));
43      assert(mat1[i] != NULL);
44    }
45      mat1=pi_mat;
46      printf("D_mat :\n" );
47      for(int i=0;i<node_num;i++)
48      {
49          for(int j=0;j<node_num;j++)
50          {
51              printf("%5d  ",D_mat[i][j]);
52          }
53          printf("\n");
54      }
55
56      printf("pi_mat: \n" );
57      for(int i=0;i<node_num;i++)
58      {
59          for(int j=0;j<node_num;j++)
60          {
61              printf("%5d  ",pi_mat[i][j]);
62          }
63          printf("\n");
64      }
65
66      //update part
67      for(int i=0;i<node_num;i++)
68      {
69          for(int j=0;j<node_num;j++)
70          {
71              mat[i][j]=MIN(D_mat[i][j],sum(D_mat[i][iter],D_mat[iter][j]));
72              if(mat[i][j]!=D_mat[i][j])
73              {
74                  pred=iter;
75                  if(mat1[iter][j]!=pred)
76                  {
77                      pred=mat1[iter][j];
78                      mat1[i][j]=pred;
79                  }
80              }
81          }
82      }
83      D_mat=mat;
84      pi_mat=mat1;
85  }
86
87  void pi_D_mat_calc_0()
88  {
89      int **mat,**mat1;
90      mat=(int **)malloc(node_num*sizeof(int *));
91      mat1=(int **)malloc(node_num*sizeof(int *));
92      D_mat=(int **)malloc(node_num*sizeof(int *));
93      pi_mat=(int **)malloc(node_num*sizeof(int *));
94    for(int i=0;i<node_num;i++)
95    {
96      mat[i]= (int *)malloc(node_num*sizeof(int));
97      assert(mat[i] != NULL);
98          mat1[i]= (int *)malloc(node_num*sizeof(int));
99      assert(mat1[i] != NULL);
```

```
100         D_mat[i]= (int *)malloc(node_num*sizeof(int));
101     assert(D_mat[i] != NULL);
102         pi_mat[i]= (int *)malloc(node_num*sizeof(int));
103     assert(pi_mat[i] != NULL);
104   }
105     for(int i=0;i<node_num;i++)
106   {
107     for(int j=0;j<node_num;j++)
108     {
109       if(i==j)
110             {
111                 mat[i][j]=0;
112                 mat1[i][j]=INF;
113             }
114             else if(adj_mat[i][j]==1)
115             {
116                 mat[i][j]=wt_mat[i][j];
117                 mat1[i][j]=i;
118             }
119             else
120             {
121                 mat[i][j]=INF;
122                 mat1[i][j]=INF;
123             }
124     }
125   }
126     D_mat=mat;
127     pi_mat=mat1;
128 }
129
130 void Floyd_Warshall()
131 {
132     pi_D_mat_calc_0();
133     //printf("Hiiii\n");
134     //printf("%d\n", node_num);
135     for(int count=0;count<node_num;count++)
136     {
137         //printf("Haaa\n" );
138         pi_D_mat_calc(count);
139     }
140 }
141 int main()
142 {
143   int start_node,end_node;
144
145   printf("Enter node_num : " );
146   scanf("%d", &node_num);
147     //adj_mat
148   adj_mat=(int **)malloc(node_num*sizeof(int *));
149   for(int i=0;i<node_num;i++)
150   {
151     adj_mat[i]= (int *)malloc(node_num*sizeof(int));
152     assert(adj_mat[i] != NULL);
153   }
154   printf("Enter adj_mat :\n" );
155   for(int i=0;i<node_num;i++)
156   {
157     for(int j=0;j<node_num;j++)
158     {
159       scanf("%d", &adj_mat[i][j]);
```

```
160       }
161     }
162       //wt_mat
163       wt_mat=(int **)malloc(node_num*sizeof(int *));
164     for(int i=0;i<node_num;i++)
165     {
166       wt_mat[i]= (int *)malloc(node_num*sizeof(int));
167       assert(wt_mat[i] != NULL);
168     }
169     printf("Enter adj_mat :\n" );
170     for(int i=0;i<node_num;i++)
171     {
172       for(int j=0;j<node_num;j++)
173       {
174         scanf("%d", &wt_mat[i][j]);
175       }
176     }
177
178       Floyd_Warshall();
179
180       printf("D_mat :\n" );
181       for(int i=0;i<node_num;i++)
182       {
183           for(int j=0;j<node_num;j++)
184           {
185               printf("%5d  ",D_mat[i][j]);
186           }
187           printf("\n");
188       }
189
190       printf("pi_mat: \n" );
191       for(int i=0;i<node_num;i++)
192       {
193           for(int j=0;j<node_num;j++)
194           {
195               printf("%5d  ",pi_mat[i][j]);
196           }
197           printf("\n");
198       }
199       return 0;
200 }
```

We will use the following adjacent and cost matrix :

```
adjacent matrix
0 1 0 0 1 0 0 0
0 0 1 0 0 1 0 0
0 0 0 1 0 0 0 1
0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 1 0 0 0 0 0
0 0 1 0 0 0 1 0
cost matrix
0 1 0 4 0 0 0 0
0 0 2 0 0 7 0 0
0 0 0 2 9 0 0 9
```

0 0 0 0 0 0 0 0
3 5 0 0 0 0 0 0
0 0 0 0 6 0 0 0
0 0 8 0 0 0 0 0
0 0 2 0 0 0 7 0

Output is:

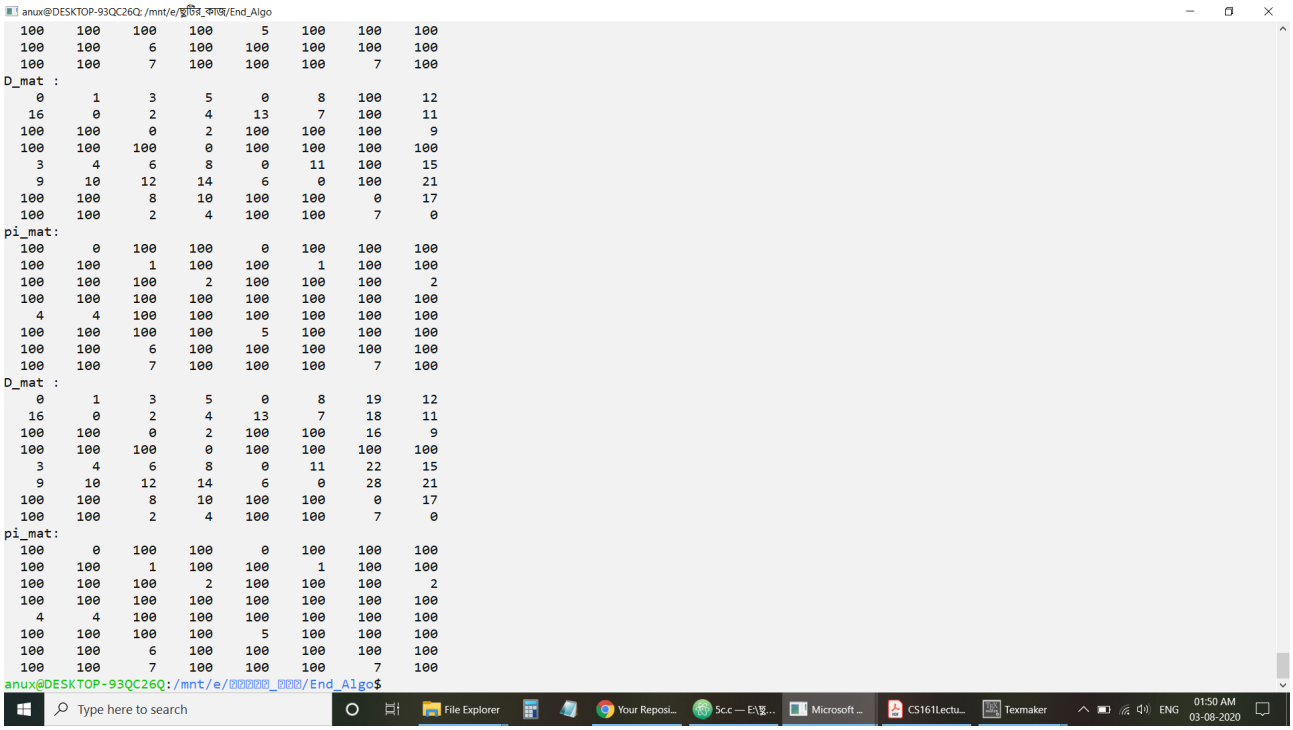

Figure 3: Floyed-Warshall algo

Detailed output we have uploaded: at https://github.com/ghosh-sarbajit

6. (a) **Largest common subsequence :** Let us consider two sequence of characters X and Y. We will say that sequence Y is an subsequence of X if Y can be obtained from X by deleting symbols.
Example : Consider X="asdtreerqoplw" and Y="ateow", here Y is subsequence of X.
Now for two strings X and Y their longest common subsequence will be defined by string Z which can be obtained by omitting symbols from X and Y.

   (b) We will solve longest common subsequence problem by using dynamic programming method.
Let us consider given strings are $X[1:n]$ and $Y[i:m]$.
Let us consider a matrix $A[0:n][0:m]$. In this matrix we will store the the length of longest common subsequence between the two given strings.Based on whether last character of two strings are common or not we can obtain the following recursive relation between elements of A.

$$A[i][j] = \begin{cases} 0, & i = 0 \vee j = 0 \\ A[i-1][j-1] + 1, & X[i] = Y[j], i, j > 0 \\ max\{A[i-1][j], A[i][j-1]\} & X[i] \neq Y[j], i, j > 0 \end{cases}$$

Based on the above algorithm for longest common subsequence is given as follows.

Algorithm 4 Length_Of_LCS(X,Y)

```
1: Initialize A[n+1][m+1]
2: A[0][j]=A[i][0] ∀i, j ∈ {1, ..., n} × {1, ..., m}
3: for k ← 1(1)n do
4:     for i ← 1(1)m do
5:         if X[i]=Y[j] then
6:             A[i][j] ← A[i-1][j-1] + 1
7:         else
8:             A[i][j] ← max{A[i-1][j], A[i][j-1]}
9:         end if
10:    end for
11: end for
12: return A
```

After finding the length of longest common subsequence we between we are giving algorithm for finding longest common subsequence string. The following algorithm finds length longest common subsequence with the help of table A.

Algorithm 5 LCS(X,Y,A)

```
1: Initialize str ← ∅ ∧ i ← n ∧ j ← m
2: while i > 0 ∧ j > 0 do
3:     if X[i] = Y[j] then
4:         str=X[i] ∘ str
5:         i ← i-1
6:         j ← j-1
7:     else if A[i][j]=A[i][j-1] then
8:         j ← j-1
9:     else
10:        i ← i-1
11:    end if
12: end while
```

C implementation of above algorithm

```c
#include <stdio.h>
#include <string.h>
#include <string.h>
#define MAX(a,b) (a>b)?a:b
int lcs(char *, char *, char *);

int main()
{
  int k=0;
  char str1[100],str2[100],str3[100];
  printf("Enter str1 : " );
  scanf("%[^\n]s",str1);
  getchar();
  //fgets(str1,99,stdin);
  printf("Enter str2 : " );
  scanf("%[^\n]s",str2);
```

```
  getchar();
  //fgets(str2,99,stdin);
  //puts(str1);
  //puts(str2);
  k=lcs(str1,str2,str3);
  printf("Length of largest common subseq is : %d\n", k);
  printf("Largest common sub-sequence is : %s\n", str3);
  return 0;
}

int lcs(char *str1,char *str2,char *str3)
{
  int m,n,length;
  m = strlen(str1) + 1;
  n = strlen(str2) + 1;
  //printf("%d %d\n", m,n);
  int arr[m][n];
  for(int i=0;i<m;i++)
  {
    for(int j=0;j<n;j++)
    {
      if(i==0 || j==0)
        arr[i][j]=0;
      else if(str1[i-1]==str2[j-1])
        arr[i][j]= 1 + arr[i-1][j-1];
      else
        arr[i][j]= MAX(arr[i-1][j],arr[i][j-1]);
    }
  }
  /*printf("\n\n" );*/
  for(int i=0;i<m;i++)
  {
    for(int j=0;j<n;j++)
    {
      printf("%5d", arr[i][j]);
    }
    printf("\n" );
  }
  length=arr[m-1][n-1] - 1;
  for(int i=m-1;i>=0;)
  {
    for(int j=n-1;j>=0;)
    {
      if(arr[i][j]==arr[i][j-1])
      {
        j--;

      }
      else if(arr[i][j]==arr[i-1][j])
      {
        i--;
      }
      else
      {
        i--;
        j--;
        str3[length]=str1[i];
        length--;
      }
    }
```

```
77      }
78      str3[arr[m-1][n-1]]='\0';
79      return  arr[m-1][n-1];
80  }
```

Figure 4: LCS example

(c)
- Algorithm Length_Of_LCS(X,Y) has 2 for loops, and with in each for loop we are doing $\mathcal{O}(1)$ amount calculation hence the complexity is $\mathcal{O}(mn)$.
- In algorithm LCS(X,Y,A) the sum m+n is gets decremented, and algorithm stops when it becomes 0. Hence runtime of this algorithm is $\mathcal{O}(m+n)$.

7. (a) A greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. **-CLRS**
Example: An examination system consisting three tier, with the rule that students who had qualified in the first tier are allowed to appear on the second tier and so on, is an example of greedy method of selection.

(b) Let, **G(V,E,W)** be an undirected weighted graph (given). Where V, E, and W are the sets of vertices, edges and edge weights respectively. $W = (w_{ij})$, and $w_{ij} = w(v_i, v_j)$, and $w : E \to \mathbb{R}$
Minimum Spanning Tree: A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. **- Wikipedia**
We will present **Kruskal's** algorithm for finding minimum spanning tree. Kruskal algorithm follows greedy approach for finding minimum spanning tree.
The algorithm uses disjoint-set data structure.
We will use three methods for this data structure, viz.
- **MakeSet(v_1):** This method of disjoint-set data structure crates a new set whose only member is v_1.

- **Union(v_1, v_2):** This method of disjoint-set data structure makes union of the dynamic sets containing v_1, v_2.
- **FindSet(v_1):** This method of disjoint-set data structure return a pointer the representative of the dynamic set containing v_1.

Algorithm 6 $Kruskal(V, E, W)$

```
1: T ← ∅                                              ▷ An empty set of edges
2: for ∀v1 ∈ V do
3:     MakeSet(v1)
4: end for
5: sort e_ij ∀e_ij ∈ E in increasing order of w_ij    ▷ e_ij is edge between (v_i, v_j), v_i, v_j ∈ V
6: for ∀e_i ∈ E ,taking in increasing order of w_ij do   ▷ greedy method used
7:     if FindSet(u) ≠ FindSet(v) then
8:         T ← T ∪ e_ij                               ▷ Which is safe for T
9:         Union(v_i, v_j)
10:    end if
11: end for
12: return T
```

(c) To prove correctness of Kruskal's algorithm we put a **loop invariant** at the beginning of each loop T should be subset of minimum spanning tree. It works as follows.

- **Initialization:** After initialization T is trivially satisfies **loop invariant**.
- **Maintenance:** We claim that the procedure follows in for loop of our algorithm, yields a safe edge for T. Also the graph will remain acyclic.
- **Termination:** Since all edges added in the for loop are safe edges, hence our algorithm will return us minimum spanning tree.

Existence of Safe Edge: It must exist since $T \subseteq \mathbb{T}$, where $\mathbb{T}$ is minimum spanning tree. Therefore $\exists\, e \in \mathbb{T}\, but\, e \notin T$. Thus e is safe for T. Now correctness of algorithm boils down to finding of an safe edge.
Before giving the procedure how find a safe edge and our claim made before we would like to give two definition.
Cut: For graph $G = (V, E)$ a cut $(V_1, V - V_1)$ is a partition of V. An edge said to be **crossing the cut** $(V_1, V - V_1)$ if its one end in V_1 and other end in $V - V_1$.
Light edge: An edge with minimum weight among all the edges which crosses a cut is called light edge.
Further a cut $(V_1, V - V_1)$ is said to be **respect a set** T, if no edges of T crosses the cut.
Now we shall give a theorem which will establish our claim.

Theorem: Let $G = (V, E)$ be a connected, undirected graph a real valued weight function $w : E \to \mathbb{R}$. Let $T \subset E$ and also subset of some minimum spanning tree. Consider $C = (V_C, E_C)$ a connected component in the forest $G_T = (V, T)$, and $e = (v_1, v_2)$ is a light weight edge, which connects C to any other components in G_T, then e is safe for T.
Proof: Given C is a connected component. Hence $(V_C, V - V_C)$ is a cut.
Let $\mathbb{T}$ is a MST, which contains T. Now if $e \notin \mathbb{T}$ there is no conflict we simply construct $\mathbb{T}' = \mathbb{T} \cup \{e\}$, which will make e safe for T, and we are done.

Now if $e \in \mathbb{T}$ and its inclusion forms a cycle, with the vertices $v_1 \, and \, v_2 \in T$. Since e belongs to the cut $(V_C, V - V_C)$ therefore $v_1 \in V_C and v_2 \in V - V_C$. Hence there is an edge $\mathbb{T} \ni$ its two vertices are in two sets of cut C. Let e' be such an edge. Here we make note that $e' \notin T$ since the cut respects T.
Now if we remove e' from $\mathbb{T}$ it will break into two components and will again become one if we join them with e.
Thus we form a new tree as $\mathbb{T}' = \mathbb{T} - e' \cup e$, and by above argument it will be a spanning tree.
Now to show that it is also minimum spanning tree, e was the light weight edge that was crossing the cut.
Hence, $w(e) \leq w(e')$.
$w(\mathbb{T}) = w(\mathbb{T}) - w(e') + w(e)) \leq w(\mathbb{T})$
Again, $w(\mathbb{T}) \leq w(\mathbb{T})$ as $\mathbb{T}$ is a minimum spanning tree.
So $\mathbb{T}'$ is also MST. To show that e is safe for T we notice that,
$T \subseteq \mathbb{T}' \, and \, T \subseteq \mathbb{T} also \, e' \notin T$. Thus $T \cup e \subseteq T'$. So e is safe for T this fact follows from T' is also minimum spanning tree. Hence the result follows. **(Proved)**
As in each step of the Kruskal algorithm we are including a safe edge each time the set T always remains acyclic. Thus Kruskal algorithm indeed provides a minimum spanning tree.

(d) Complexity of Kruskal algorithm on the assumption that it has been implemented using disjoint set forest.

- In line 5 there are $\mathcal{O}(|E|log(|E|))$ many operations.
- In for loop $\mathcal{O}(|E|)$ many FindSet and Union operation. $\mathcal{O}(|V|)$ many MakeSet operation. Therefore total $\mathcal{O}((|V|+|E|)\alpha(|V|))$ many operations. Where $\alpha(|V|)$ is a small growing function of $|V|$.
- Hence complexity is $\mathcal{O}(|E|log(|E|))$, or $\mathcal{O}(|E|log(|V|))$. [By remembering $|E| < |V|^2$]

8. (a) We will start by giving some necessary definition. Let $G = (V, E)$ be an directed graph and $c : E \to \mathbb{R}^+$ be a real valued cost function. We choose two $s, t \in V$ as the source and sink vertex.

Maximum flow problem:

Definition 1. Flow: A flow is a function $f : E \to \mathbb{R}^+$ with the following constraints
1. $\forall \, (v_1, v_2) \in E, \; 0 \leq f(v_1, v_2) \leq c(v_1, v_2)$ [Known as capacity constraint]
2. $\forall \, v \in V - \{s, t\}$

$$\sum_{x \in N_{in}(v)} f(x, v) = \sum_{x \in N_{out}(v)} f(v, y)$$

[Known as conservation constraint]

Value of the flow is,

$$|f| = \sum_{x \in N_{out}(v)} f(v, y) - \sum_{x \in N_{in}(v)} f(x, v)$$

in a network where in source node there are both in and out edges. Max flow problem is to maximize $|f|$.

Minimum cut problem:

Definition 2. s-t cut: Here we will partition set V into two sets $SandV \ni V = S \cup T$ where S, T are disjoint set of vertices, and $s \in S \wedge t \in T$. Now cost of cut is defined by

$$\big\|S,T\big\| = \sum_{v_1 \in S, v_2 \in T} c(v_1, v_2)$$

.

An Assumption: We impose a condition on G. For $v_1, v_2 \in G.V$ both edges $(v_1, v_2) and (v_2, v_1)$ are not in E. If they are we can easily transform the given graph introducing another new vertex v' and edges $(v_1, v') and (v', v_1)$.

Definition 3. Residual capacity: Here our objective is to improve the flow. In order to that we would like to introduce residual flow as $c_f : V \times V \to \mathbb{R}^+$

$$c_f(v_1, v_2) = \begin{cases} c(v_1, v_2) - f(v_1, v_2), & if (v_1, v_2) \in E \\ f(v_1, v_2), & if (v_2, v_1) \in E \\ 0, & else \end{cases}$$

Definition 4. Residual graph: We define G_f to be a residual network wrt. a flow f, where $G_f.V = G.V$ and $(v_1, v_2) \in G_F.E$ if $c_f(v_1, v_2) > 0$

Now we shall give three lemma which will build the very foundation of Ford-Fulkerson's algorithm for finding the minimum cut which will maximize the flow.

Lemma 1. *For many flow f and a cut (S, T) of G, we have $|f| \leq \big\|S,T\big\|$*

Lemma 2. *For a graph G and a given flow f, if sink node is not reachable from source node then in its residual graph G_f then flow f is maximum.*

Lemma 3. *For a graph G and a given flow f, if sink node is reachable from source node then in its residual graph G_f then flow f is not maximum. Moreover $\exists\, f'$ another flow $\ni |f| \leq |f'|$.*

We will give the proof Lemma 3, as part of question question 8.c. Now with the above introduced concept we can now present Ford-Fulkerson's algorithm.

Algorithm 7 $FordFulkerson(G, s, t)$

```
Ini f ← 0
for ∀e ∈ E do
    f(e) ← 0                                              ▷ e = (v1, v2)
end for
Run any path finding algorithm to find a path from p : s ⇝ t in residual graph G_f
while ∃p : s ⇝ t in G_f do
    c_f(e) ← min{c_f(e) : e ∈ p}
    for ∀ e ∈ P do
        if e = (v1, v2 ∈ E then                           ▷ Update flow wrt residual network
            f(v1, v2) ← f(v1, v2) + c_f(p)
        else
            f(v2, v1) ← f(v2, v1) − c_f(p)
        end if
    end for
end while
return f
```

(b) Not attempted.

(c) Correctness of Ford-Fulkerson theorem follows from Lemma 2 and Lemma 3. Here we will give proof of **Lemma 3**.

Proof. Let $p : s \rightsquigarrow t$ be a simple path in G_f and $F = min_i\{c(e_i) : e_i \in p\}$ and $e_i = (v_1, v_2)$. Based on above we define a new flow as follows,

$$f'(v_1, v_2) = \begin{cases} f(v_1, v_2) + F, & if (v_1, v_2) \in p \\ f(v_1, v_2) - F, & if (v_2, v_1) \in p \\ f(v_1, v_2), & else \end{cases}$$

Now we will show that f' is indeed a flow,to prove **capacity constraints** are satisfied we notice,

- If $(v_1, v_2) \in P$ then $0 \leq f(v_1, v_2) + F \leq f(v_1, v_2) + c_f(v_1, v_2) = f(v_1, v_2) + c(v_1, v_2) - f(v_1, v_2) = c(v_1, v_2)$
- if $(v_2, v_1) \in p$ then $f(v_1, v_2) - F \leq f(v_1, v_2) \leq c(v_1, v_2)$ and $f(v_1, v_2) - F \geq f(v_1, v_2) - c_f(v_1, v_2) = 0$
- Else flow is from original flow

To prove **conservation constraints** are satisfied,

- we recall every simple path p, uses 0 or two edges on V. If p uses 0 edges on on V the flow value remains intact while going from $f \rightarrow f'$. If p uses two edges of $v \in V$ viz. $(v_1, v)\, and\, (v, v_2)$ which are incident to v. Here we consider the following cases [Note that edges appearers in opposite direction in residual network]
- If $(v_1, v)\, and\, (v, v_2)$ are both in same direction, both incoming and outgoing flow in v if incremented by F.
- If $(v_1, v)\, and\, (v, v_2)$ both are in opposite direction , both incoming and outgoing flow in v if decreased by F.
- If $(v_1, v)\, and\, (v, v_2)$ does not have same direction, then in either cases the flow does not change as $F - F = 0$.

At the end we notice that in f' we have increased the flow by F. Its follows from the fact, for $e \in p$, if $e \in E$ the outward flow increased by F, and if $-e \in E$ outward flow decreased by F. But again from our definition of residual network $F > 0$, hence we get $|f'| > |f|$. □

Complexity: Complexity of the above algorithm is given by $\mathcal{O}(|E||f'|)$

9. (a) THe idea of 3SAT and vertex cover problem is as follows:

Definition 5. 3SAT: Consider three Boolean variable viz. x, y, z. We define a **literal** as a Boolean variable and its negation. We also define a **clause** as combination of several literals connected with $\vee$, e.g. a clause could be $x \wedge \bar{y}$. We will say that a clause is in **conjunctive normal form** or in **cnf form** if several clause is connected by $\wedge$.
Now we consider a boolean function ϕ in three boolean variable which is in **cnf form** is given **3SAT problem** is to check whether there is an combination of (x, y, x) which satisfies to given Boolean function. Since clauses in the Boolean function is connected by $\wedge$ operator, equivalently we may say that we have to find a combination of (x, y, x) which satisfy all the clauses in the given Boolean function simultaneously.

Definition 6. Vertex cover problem: Let us consider an undirected graph $G = (V, E)$. A of G is a set $V' \subseteq V$ such that each edge of G touches one of these nodes. The vertex cover problem, is given a graph G and a number k whether there is a vertex cover of size k in G.

(b) Here we have to prove the **vertex cover problem** is **NP-complete on the assumption that 3SAT problem is NP-complete**. Before giving the proof we would like to give, some definitions.

Definition 7. NP: Is the set of languages, whose solution can be guessed, and that guess can be verified by some "Turin machine" in polynomial time.

Definition 8. NP-Hard: Let L be an given language. L will be NP-Hard if $\forall\, L' \in NP\ L' \leq_P L$, i.e L' is reducible to L in polynomial time.

Definition 9. NP-Complete: Let L be an given language. We will say that given language is NP-Complete if,

- $L \in NP$
- L is NP hard.

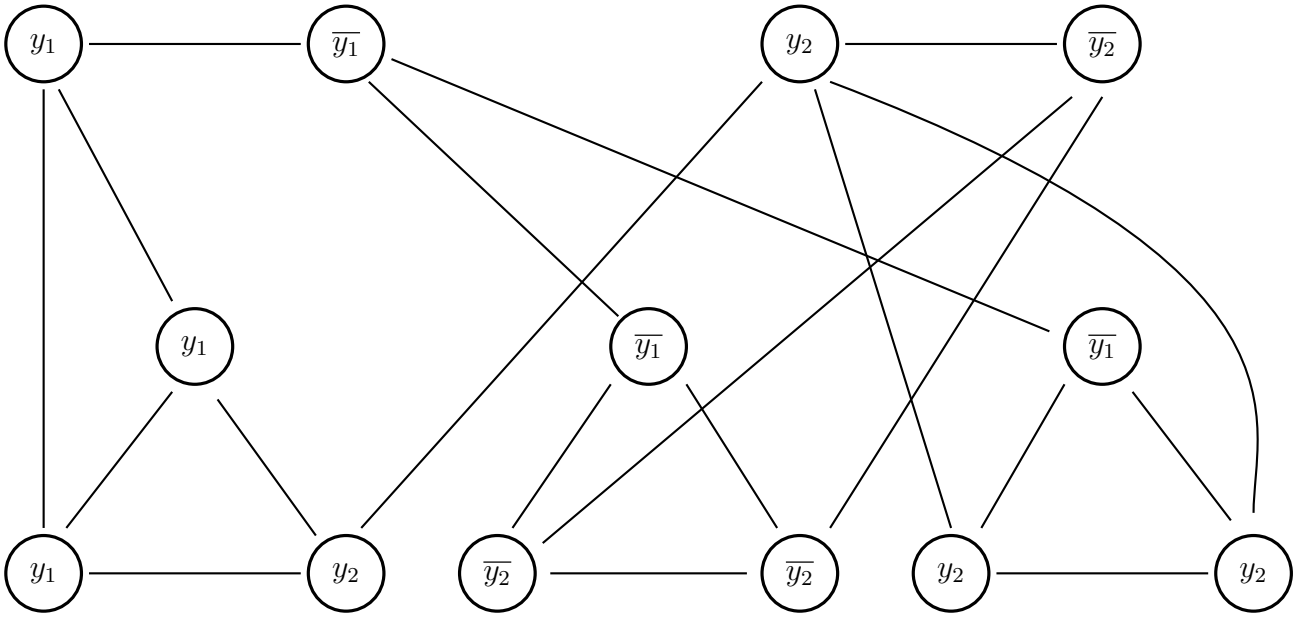

Figure 5: Vertex cover is NP-Complete

Now we will move on to the proof

Proof. Consider G an undirected graph. k is a given positive integer. Let us define the set,

$$VCOV = \{< G, k > | \textit{undirected graph } G \textit{ has a } k \textit{ many node vertex cover}\}$$

. Proving **vertex cover** is in NP is easy. As a vertex cover of size k will work as certificate.

To prove it is NP-hard we will show a map reduction from **3SAT** to **Vertex cover**. The reduction focuses on converting a 3cnf formula into a graph and an inter G and k respectively, in such a way that whenever there is a vertex cover of k in G the 3cnf

Boolean formula will be satisfied.

Let us consider following Boolean function,

$$\psi = (y_1 \vee y_1 \vee y_2) \wedge (\bar{y_1} \vee \bar{y_2} \vee \bar{y_2}) \wedge (\bar{y_1} \vee y_2 \vee y_2)$$

.

Now we set-up **gadget for variable** as follows for each variable y in ψ we draw two edges into gadget. We level the two nodes in this gadget as y and $\bar{y}$. If the node is included in vertex cover y will be TRUE else FALSE.

To set-up **gadget for clause** we do as follows. We combine three interconnected nodes to make a gadget, and connect to **variable gadget** whenever the label matches.

Now if ψ be an δ variable λ clause Boolean function then, total number of nodes in our constructed graph will be $2\delta + 3\lambda$.

$$Let,\ k := 2\delta + 3\lambda$$

. Now we will show that our given reduction works,

For that we assume that ψ is satisfiable, so we put the nodes of **variable gadget** corresponding to the true literals in assigned vertex cover and we put one true literal from every clause gadget into vertex cover. This makes cardinality of our vertex cover precisely k.

Hence 3SAT satisfiability $\implies$ Vertex Cover.

Again on the contrary we assume that G has a vertex cover of size k. Now this cover must contain one node from every **variable gadget** and two from every **clause gadget**, as it has to cover our graph both consisting of both type of gadgets. In order to make ψ satisfiable we assign TRUE to the corresponding literals.

Hence Vertex Cover $\implies$ 3SAT satisfiability.

This completes the proof.

□

(c) Not attempted