**Name : Riys Sajanlal Gupta**

**Class : SYCS**

**Rollno : 2011**

**Subject : Combinatorics and Graph Theory**

**Practical no 1**

**Aim :** Write a Python program on strings.

**Source code:**

string1 = "Ramniranjan college"

print("Initial string")

print(string1)

print("\nFirst character of string 1 : ")

print(string1[1])

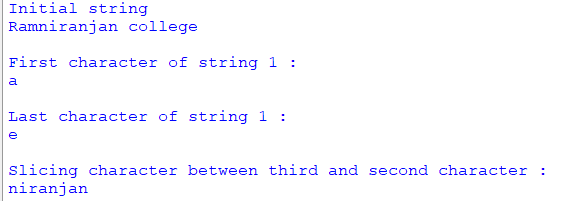
print("\nLast character of string 1 : ")

print(string1[-1])

print("\nSlicing character between third and second character :")

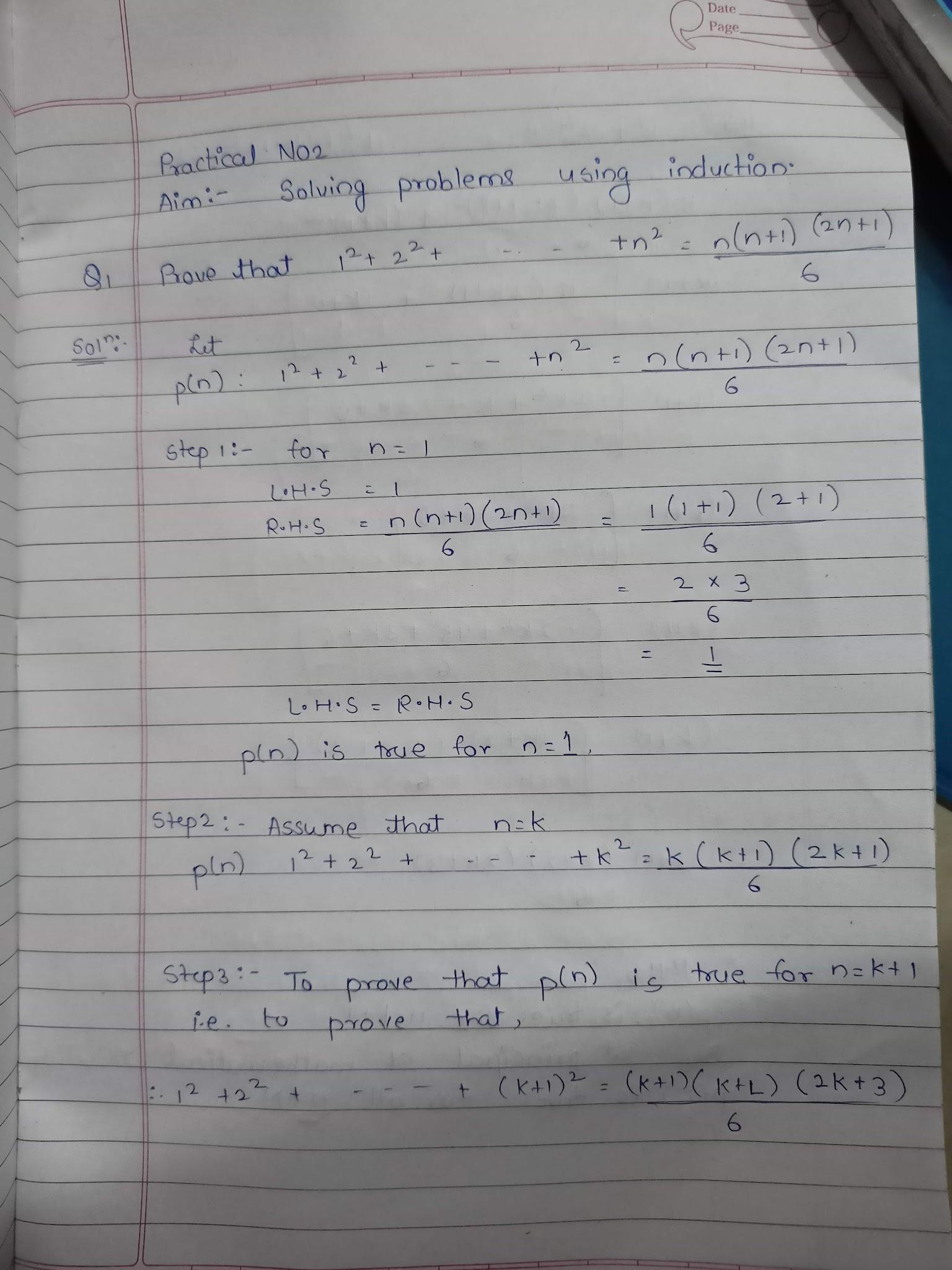
print(string1[3:12])

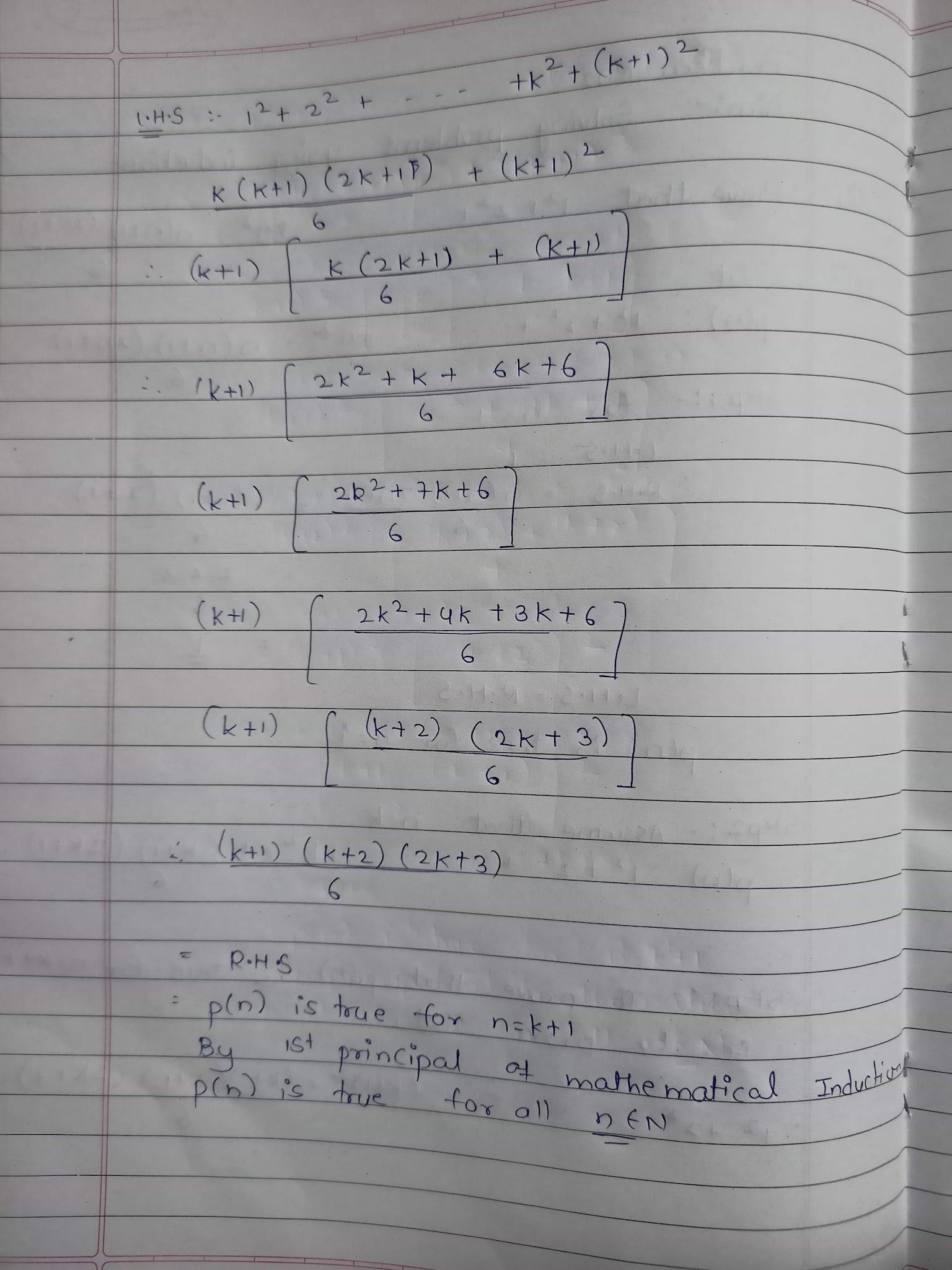
**Output:**

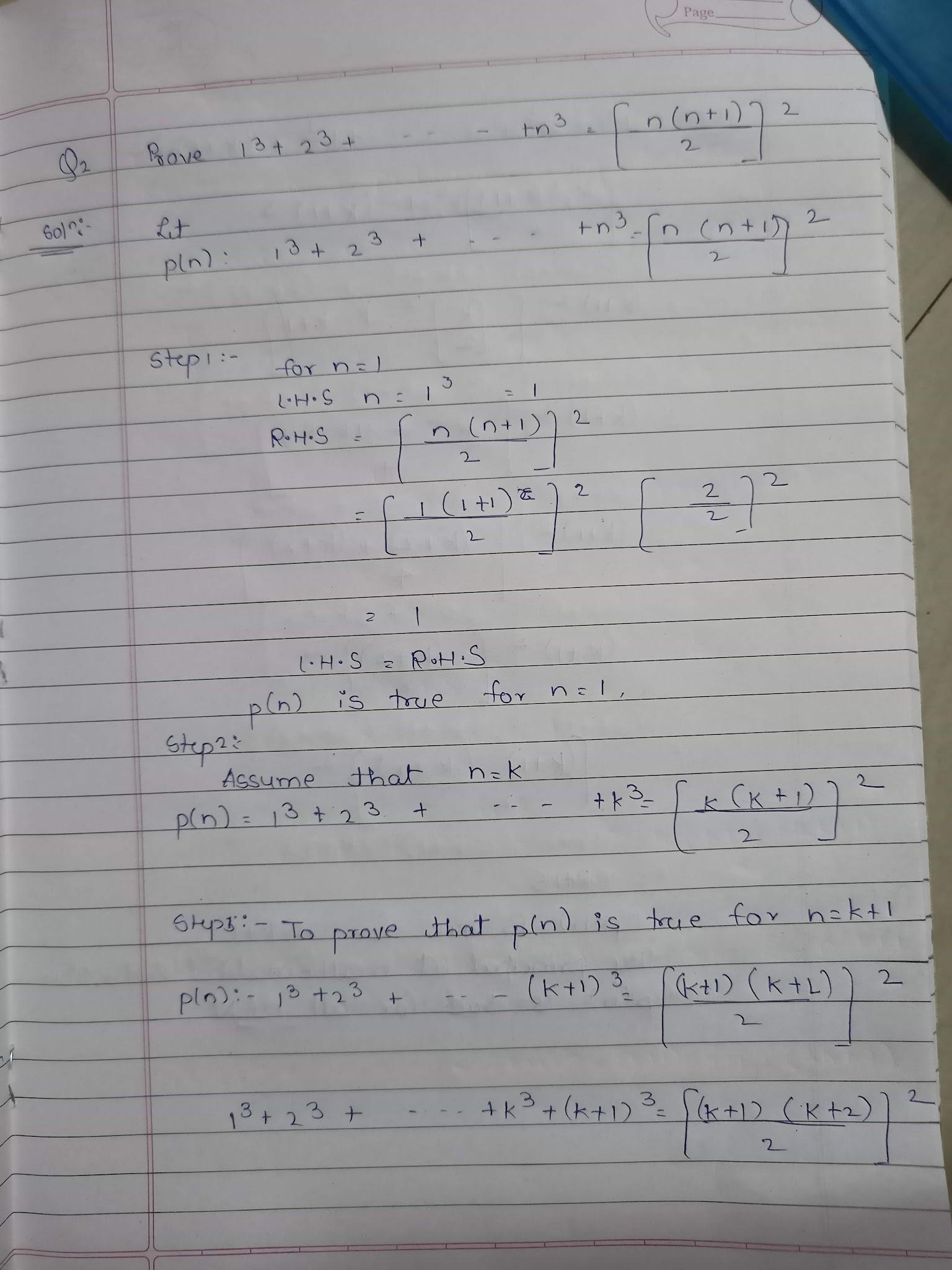


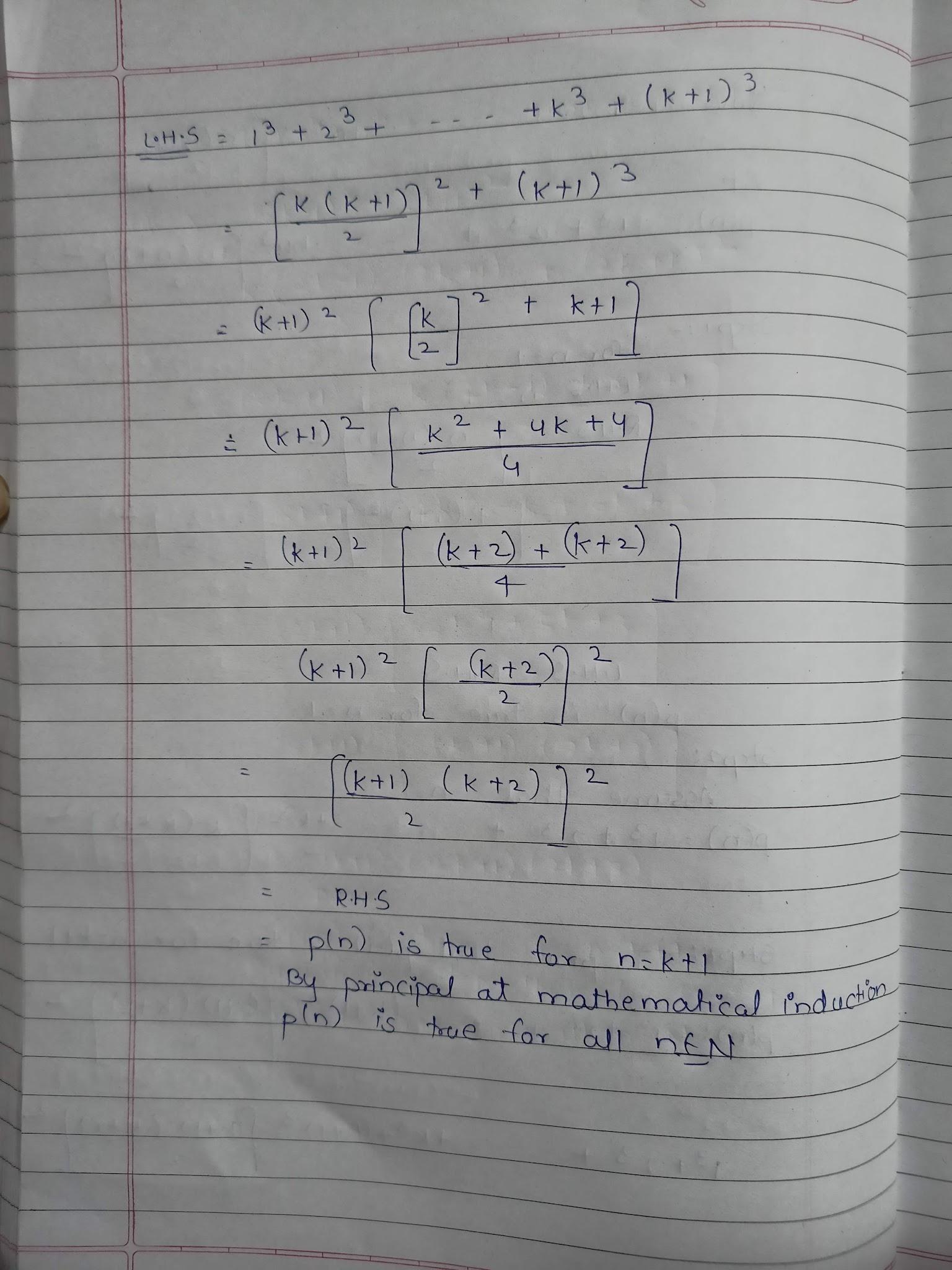
**Practical no 2**

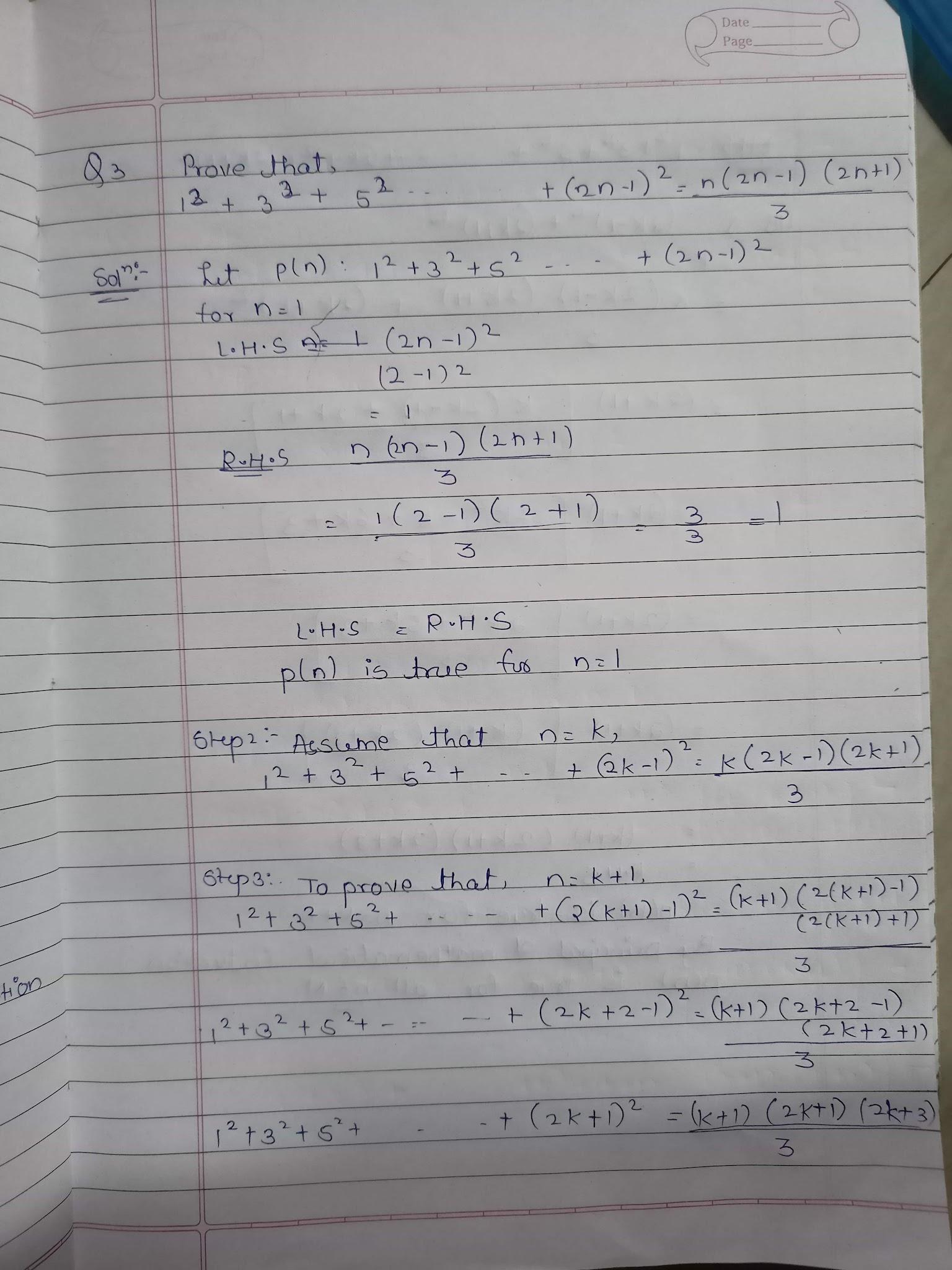
**Aim :** Solving problems using induction.

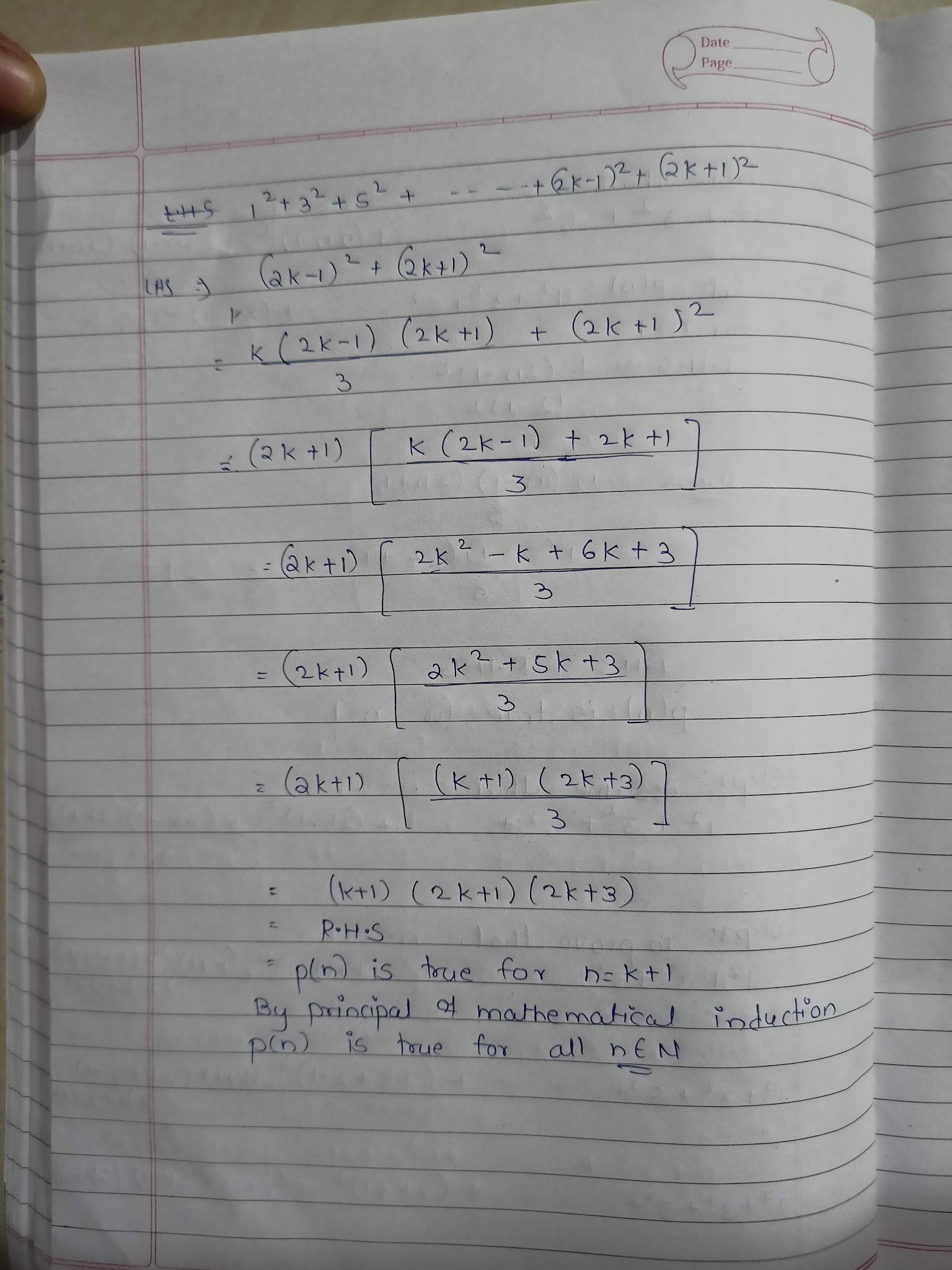
****

****

****

****

****

****

**Practical no 3**

**Aim :** Write a Python Program on Eulerian graphs.

Eulerian Tour

Recall, from section 1.6, that a trail in a graph G Is a walk in G in which the edges are distinct, i.e. no edge of G appears in the trail more than once.

A trail in G is called an Euler trail if it includes every edge of G.

Thus a trail is Euler if each edge of G is in the trail exactly once.

A tour of G is a closed walk of G which includes every edge of G at least once.

An Euler tour of G is a tour of G is a tour which includes each edge of G exactly once.

Thus an Euler tour is just a closed Euler trail.

A graph G is called Eulerian or Euler if it has an Euler tour.

**Source code:**

def find\_eulerian\_tour(graph):

tour=[]

start\_vertex = graph[0][0]

tour.append(start\_vertex)

while len(graph)>0:

current\_vertex = tour[len(tour)-1]

for edge in graph:

if current\_vertex in edge:

if edge[0] == current\_vertex:

current\_vertex =edge[1]

elif edge[1] == current\_vertex:

current\_vertex = edge[0]

else:

return false

graph.remove(edge)

tour.append(current\_vertex)

break

return tour

graph = [(1,2),(2,3),(3,1)]

print(find\_eulerian\_tour(graph))

**Output:**

****

**Practical no 5**

**Aim :** Solving problems using Kruskal’s Algorithm.

Source code:

class Graph:

def \_\_init\_\_(self,vertices):

self.V = vertices

self.graph=[]

def add\_edge(self, u,v,w):

self.graph.append([u,v,w])

def find(self,parent,i):

if parent[i] == i:

return i

return self.find(parent, parent[i])

def apply\_union(self, parent,rank,x,y):

xroot = self.find(parent,x)

yroot = self.find(parent,y)

if rank[xroot] < rank[yroot]:

parent[xroot]=yroot

elif rank[xroot] > rank[yroot]:

parent[xroot]=yroot

else:

parent[xroot]=yroot

rank[xroot]+=1

def kruskal\_algo(self):

result = []

i,e = 0,0

self.graph = sorted(self.graph, key=lambda item : item[2])

parent = []

rank = []

for node in range(self.V):

parent.append(node)

rank.append(0)

while e < self.V-1:

u,v,w = self.graph[i]

i=i+1

x=self.find(parent,u)

y=self.find(parent,v)

if x!= y:

e=e+1

result.append([u,v,w])

self.apply\_union(parent, rank,x,y)

for u,v,weight in result:

print("%d - %d : %d" %(u,v,weight))

g = Graph(6)

g.add\_edge(0,1,4)

g.add\_edge(0,2,4)

g.add\_edge(1,2,2)

g.add\_edge(1,1,4)

g.add\_edge(1,0,4)

g.add\_edge(2,0,4)

g.add\_edge(2,1,2)

g.add\_edge(2,3,3)

g.add\_edge(2,5,2)

g.add\_edge(2,4,4)

g.add\_edge(3,2,3)

g.add\_edge(3,4,3)

g.add\_edge(4,2,4)

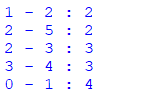
g.add\_edge(4,3,3)

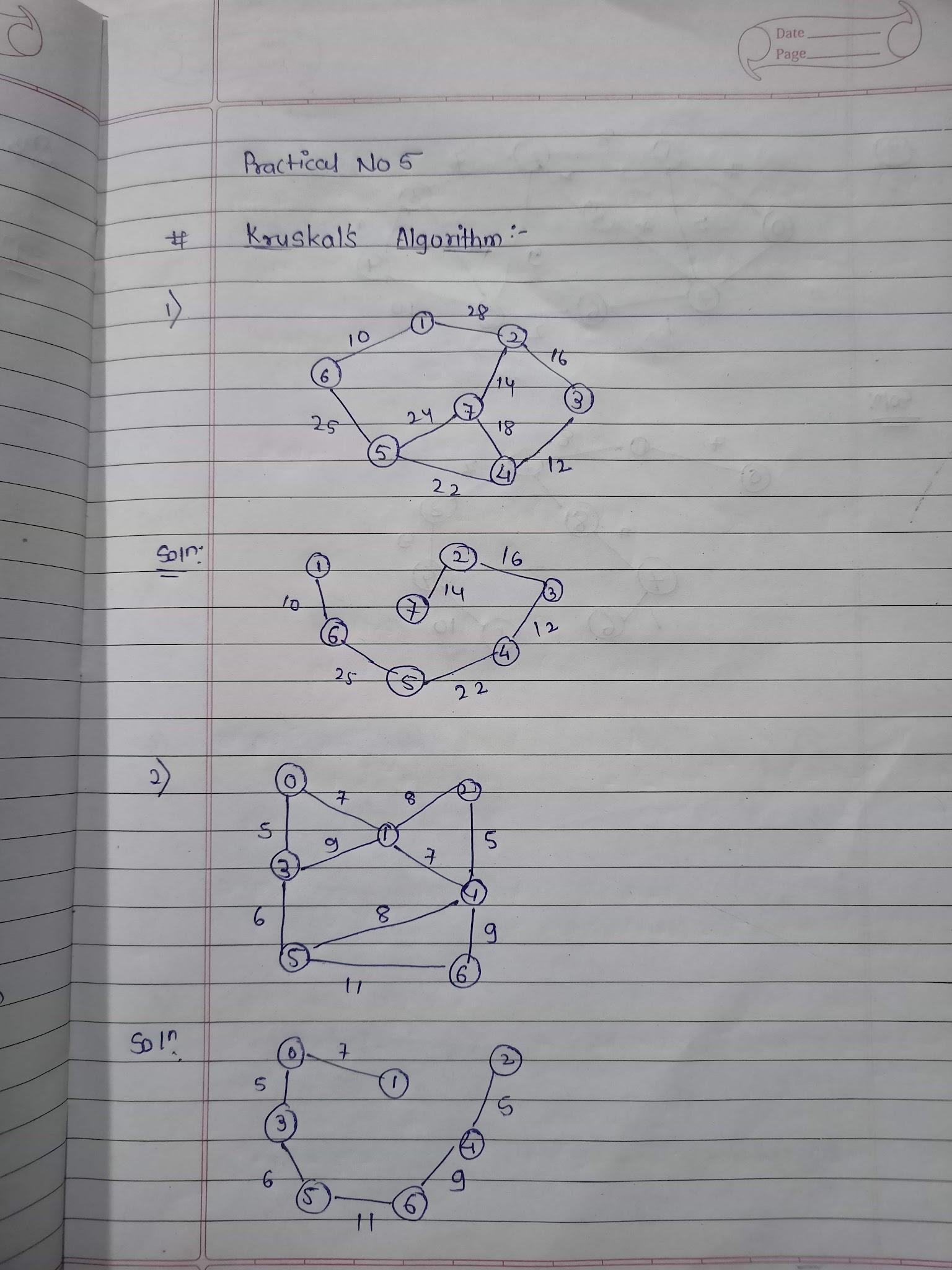
g.add\_edge(5,2,2)

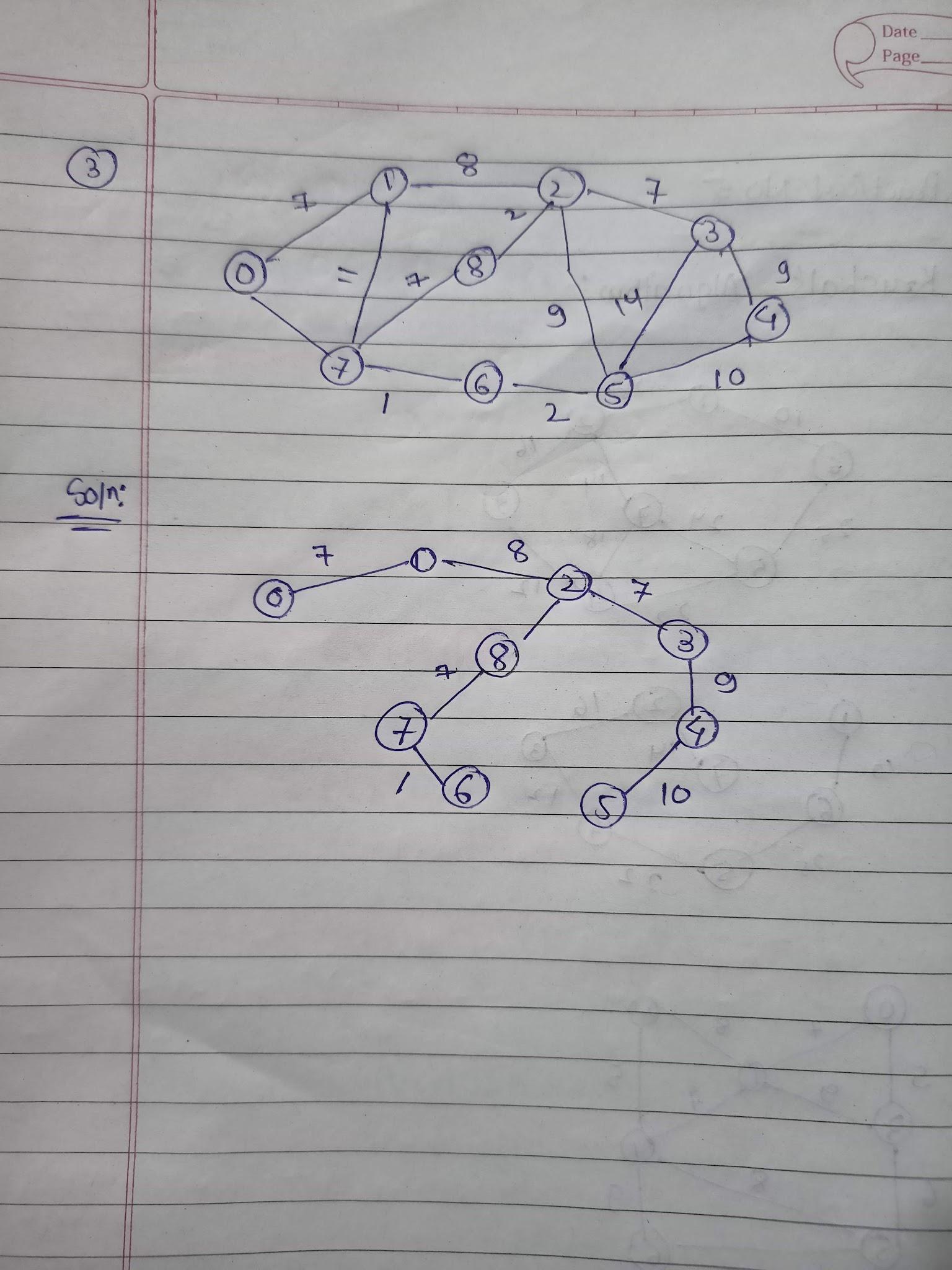
g.add\_edge(5,4,3)

g.kruskal\_algo()

**Output:**

****

****

****

**Practical no 6**

**Aim :** Solving problems using Prim’s Algorithm.

**Source code:**

INF = 9999999

V = 5

G = [[0,9,75,0,0],

[9,0,95,19,42],

[75,95,0,51,66],

[0,19,51,0,31],

[0,42,66,31,0]]

selected = [0,0,0,0,0]

no\_edge = 0

selected[0] = True

print("Edge : Weight\n")

while(no\_edge < V-1):

minimum = INF

x = 0

y = 0

for i in range(V):

if selected[i]:

for j in range(V):

if((not selected[j]) and G[i][j]):

if minimum > G[i][j]:

minimum = G[i][j]

x = i

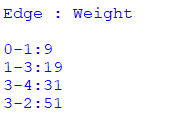
y = j

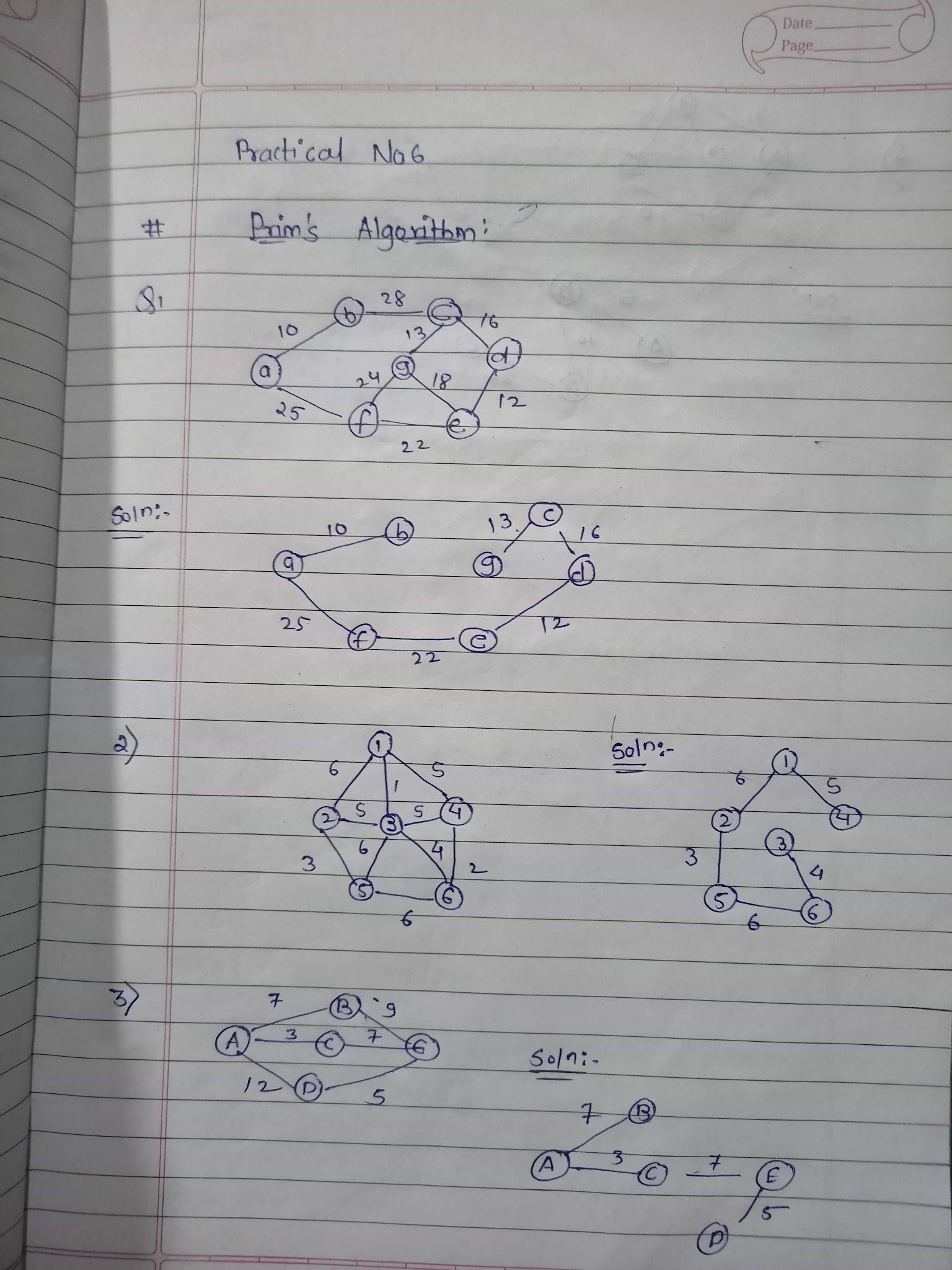
print(str(x) + "-" + str(y) + ":" + str(G[x][y]))

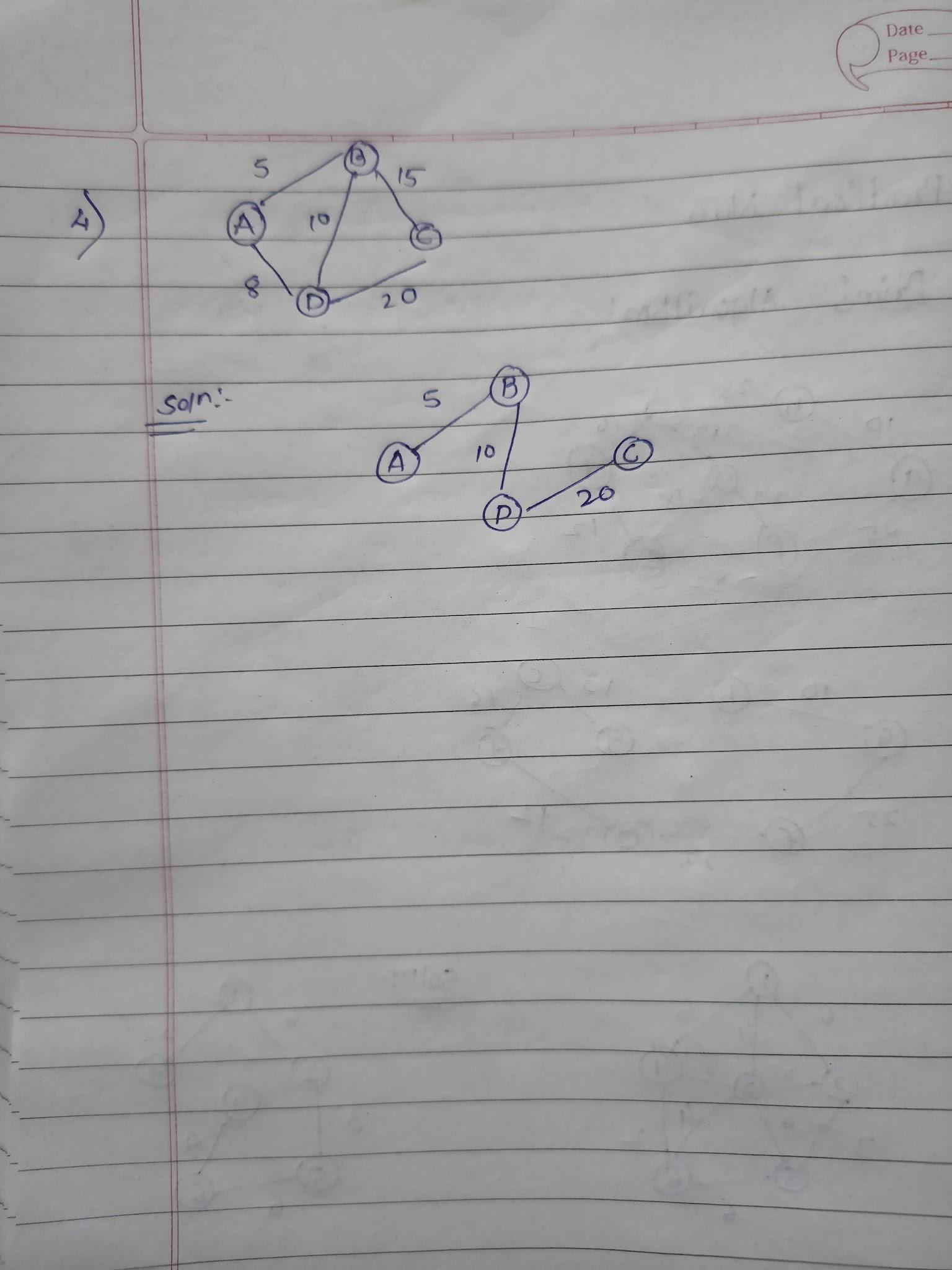
selected[y] = True

no\_edge += 1

**Output:**

****

****

****

**Practical no 7**

**Aim :** Solving problems on Dijkstra’s Algorithm.

**Source code:**

graph = {

'A':['B','C'],

'B':['D','E'],

'C':['F'],

'D':[],

'E':['F'],

'F':[],

}

visited=[]#List to keep track of visited nodes

queue=[]

def bfs(visited,graph,node):

visited.append(node)

queue.append(node)

while queue:

s=queue.pop(0)

print(s)

for neighbour in graph[s]:

if neighbour not in visited:

visited.append(neighbour)

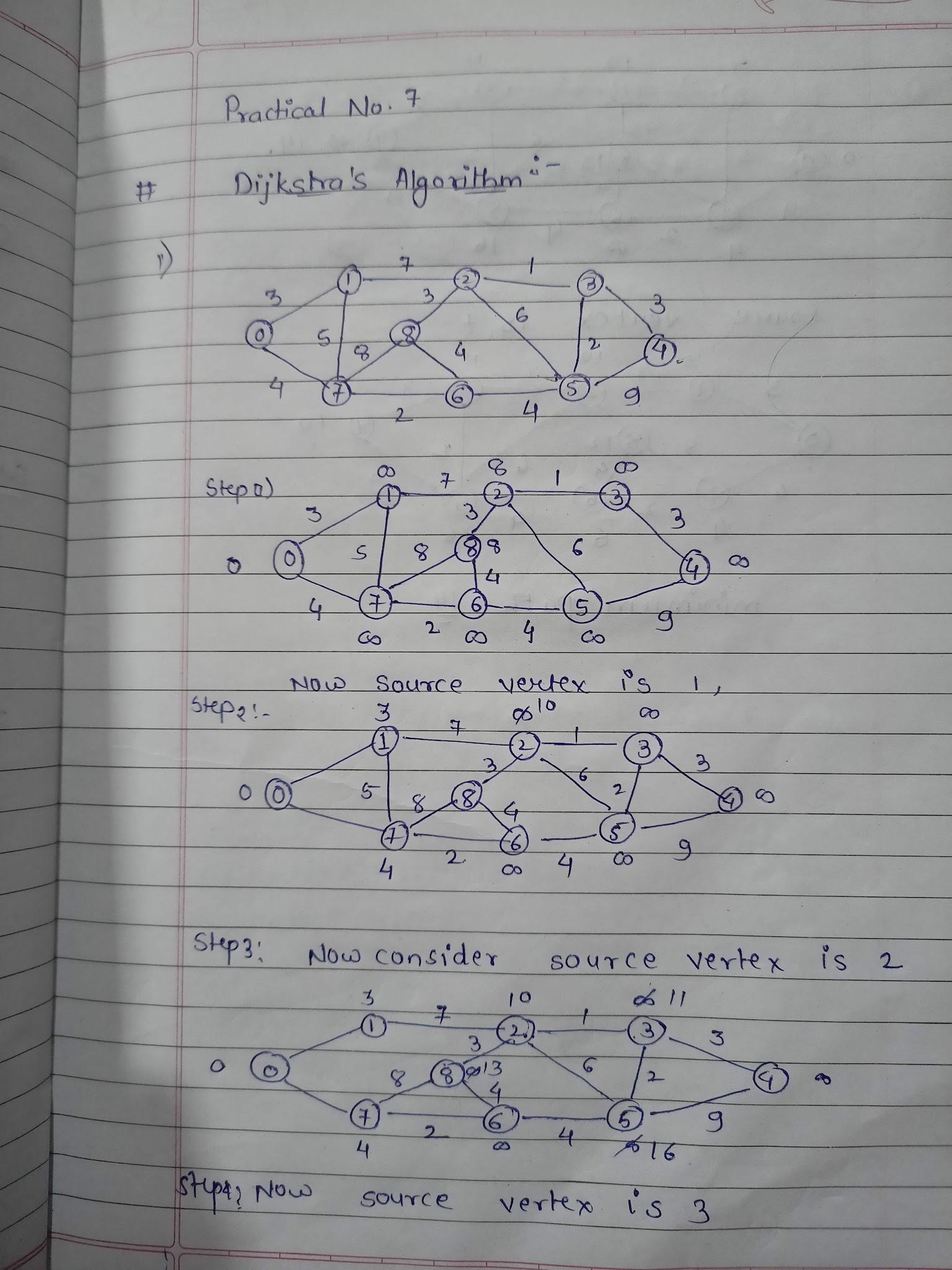
queue.append(neighbour)

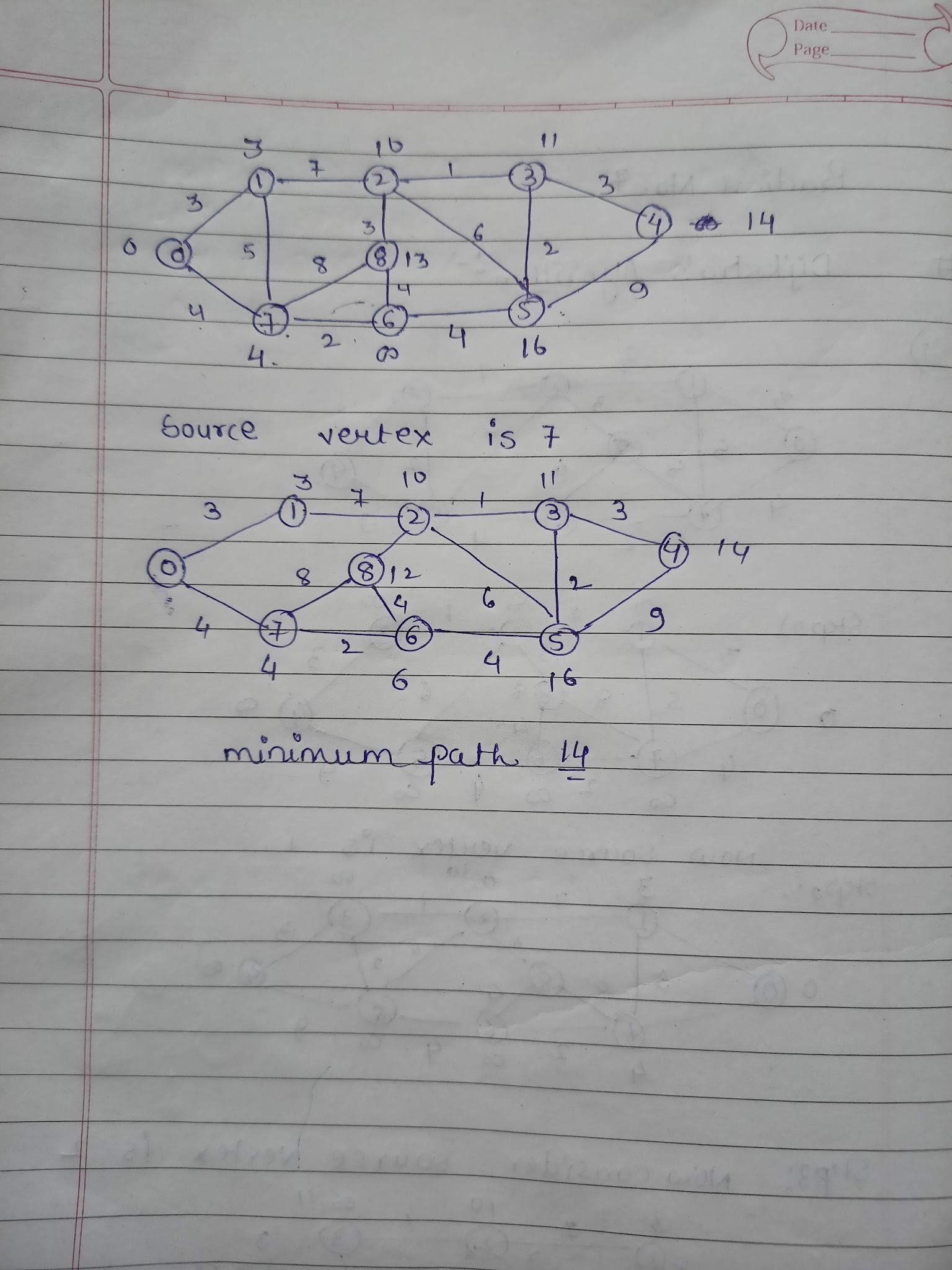
#Driver code

bfs(visited,graph,'A'**)**

**Output:**

****

****

****

**Practical no 8**

**Aim :** Solving problems on DFS Graph Traversal.

**Description:**

**DFS:-** (Depth First Search) is an algorithm for searching a graph or tree data structure. Tree algorithm starts at the root (top) node a tree and goes as far as it can down a given branch (path), then backtracks until it finds an unexplored path, and then explore entire graph has been explore.

DFS is also used in tree – traversal algorithms, also known as tree searches, which have applications in the traveling – salesman problem and the Ford – Fulkerson algorithm.

There are three different, strategies for implementing

DFS : pre-order, in-order, and post-order.

**Pre-orde**r:- DFS works by visiting the current node and successively moving to the left until a leaf is reached, visiting each node on the way there. Once there are no more children on the left of a node, the children on the right are visited. This is the most standard DFS algorithm.

Instead of visiting each node as it traverses down a tree, as **in-order** algorithm finds the leftmost node in the tree, visits that node, and subsequently visits the parent of that node. It then goes to the child on the right and finds the next leftmost node in the tree to visit.

A **post-order** strategy works by visiting the leftmost leaf in the tree, then going up to the parent and down the second leftmost leaf in the same branch, and so on until parent is the last node to be visited within a branch, this type of algorithm prioritizes the processing of leaves before root in case goal lies at the end of the tree.

**It Employs the Following Rules :**

(1)Visits the Adjacent unvisited vertex, Mark it as Visited. Display it. Push it in a stack.

(2)If no Adjacent vertex is found, pop up a vertex from the stack. (It will Pop Up all the vertices from the stack, which do not have adjacent vertices.)

(3)Repeat Rule 1 and Rule 2 until the stack is empty.

**NOTE :** Relative order of visits i.e. left-right-root

**Source code :**

graph = {

'5':['3','7'],

'3':['2','4'],

'7':['8'],

'2':[],

'4':['8'],

'8':[]

}

visited = set()

def dfs(visited, graph, node):

if node not in visited:

print(node)

visited.add(node)

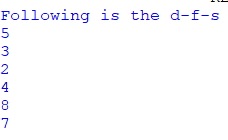
for neighbour in graph[node]:

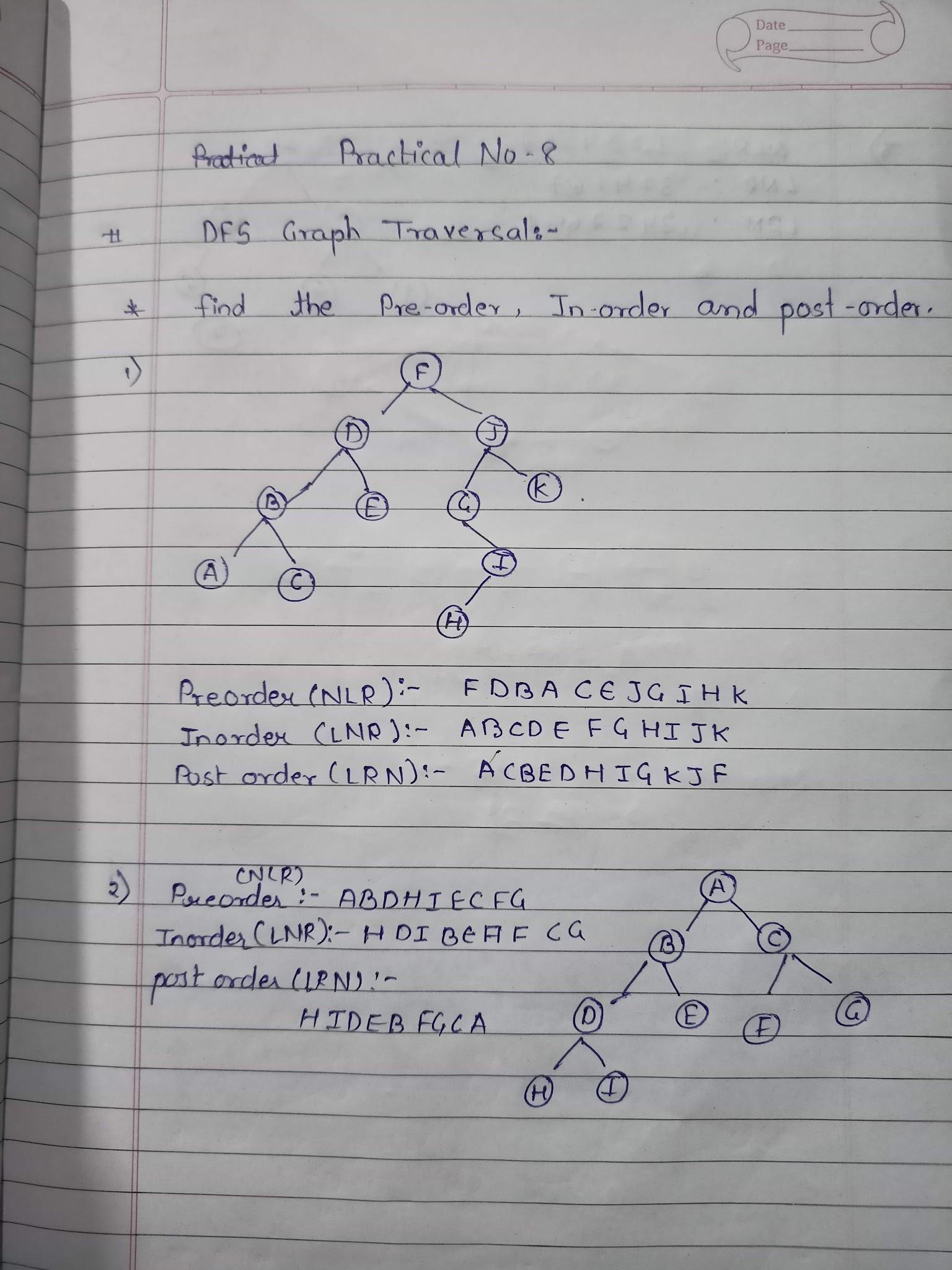
dfs(visited, graph, neighbour)

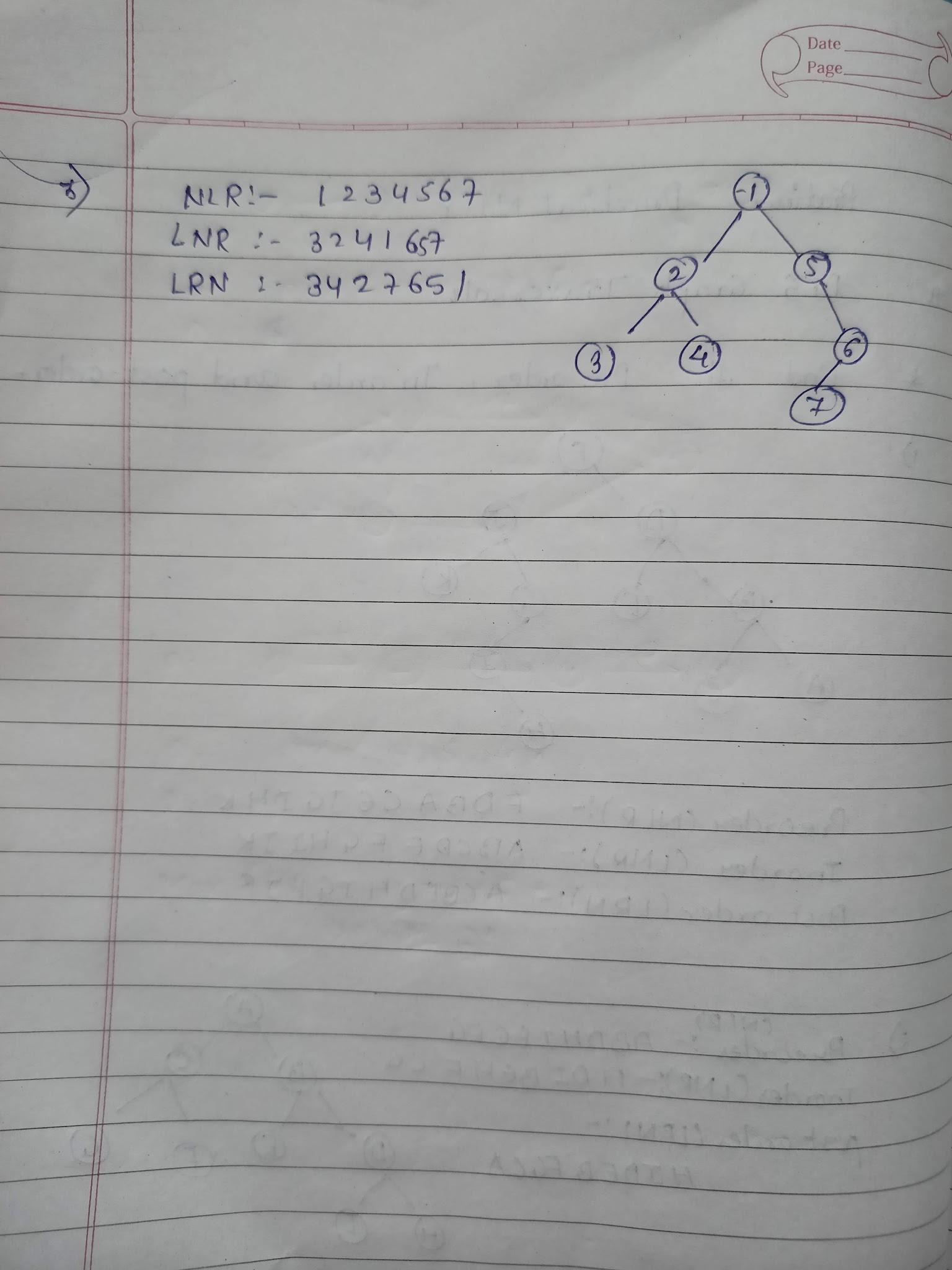
print("Following is the d-f-s")

dfs(visited, graph, '5')

**Output:**

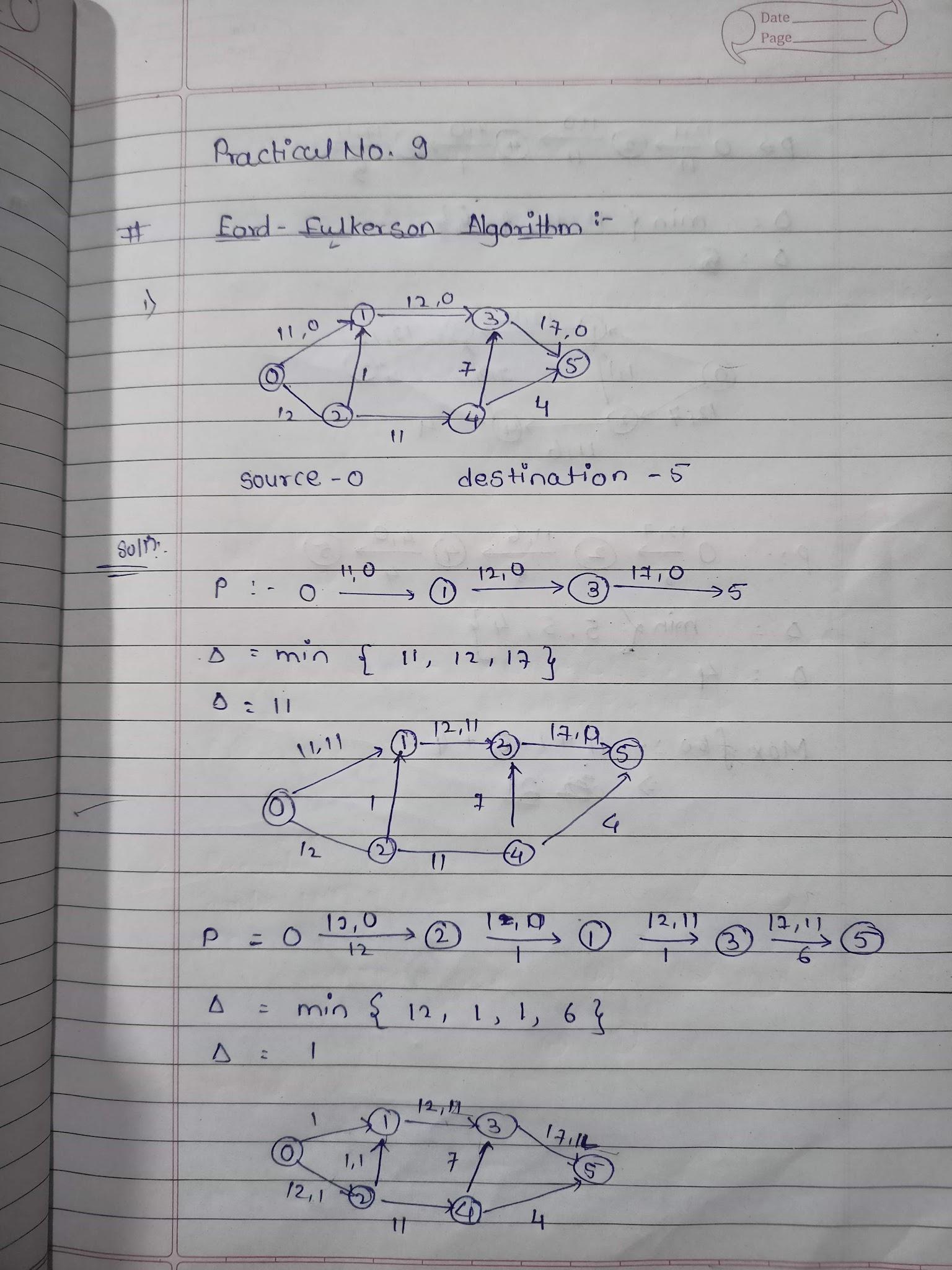


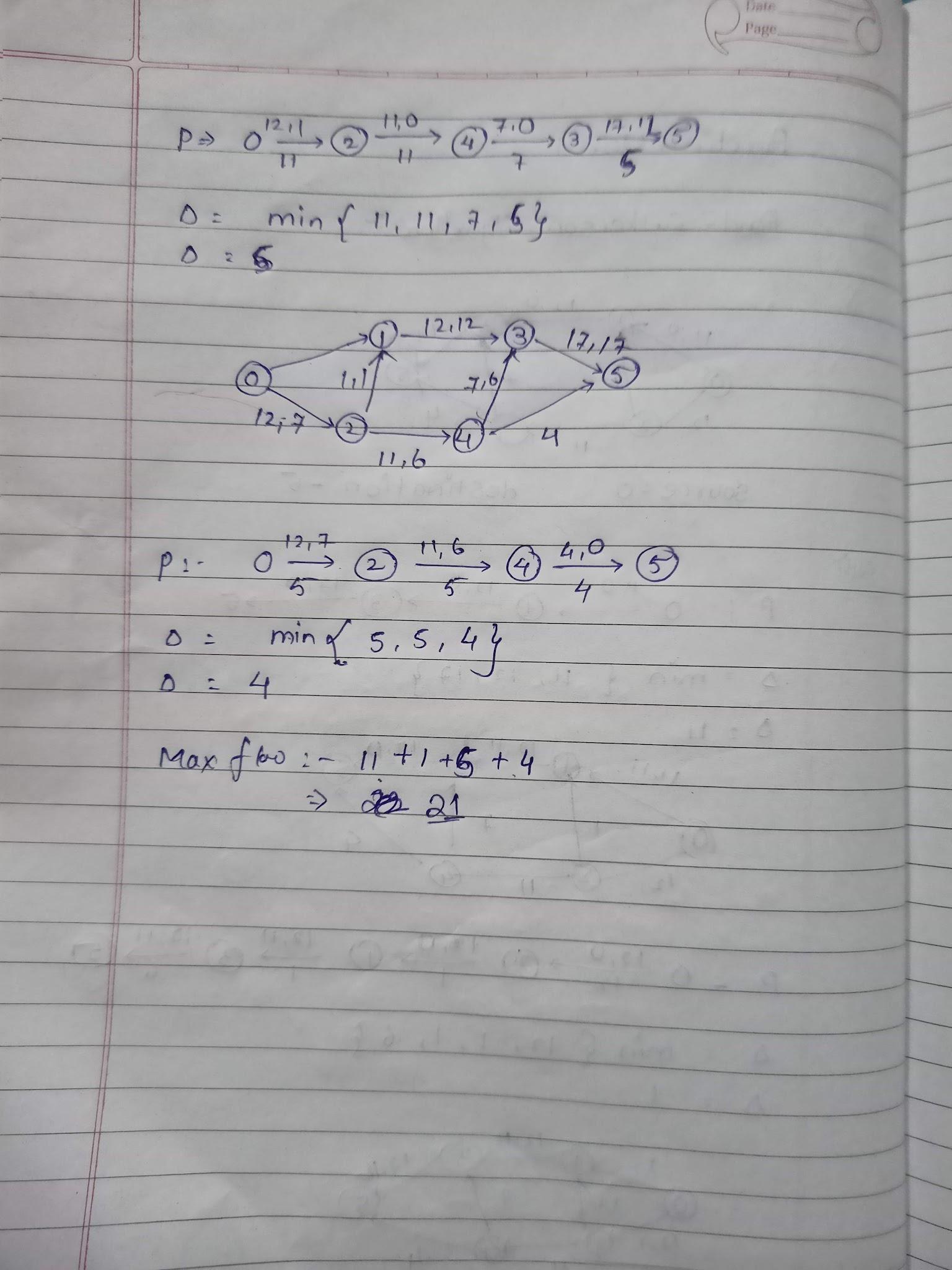
****

****

**Practical no 9**

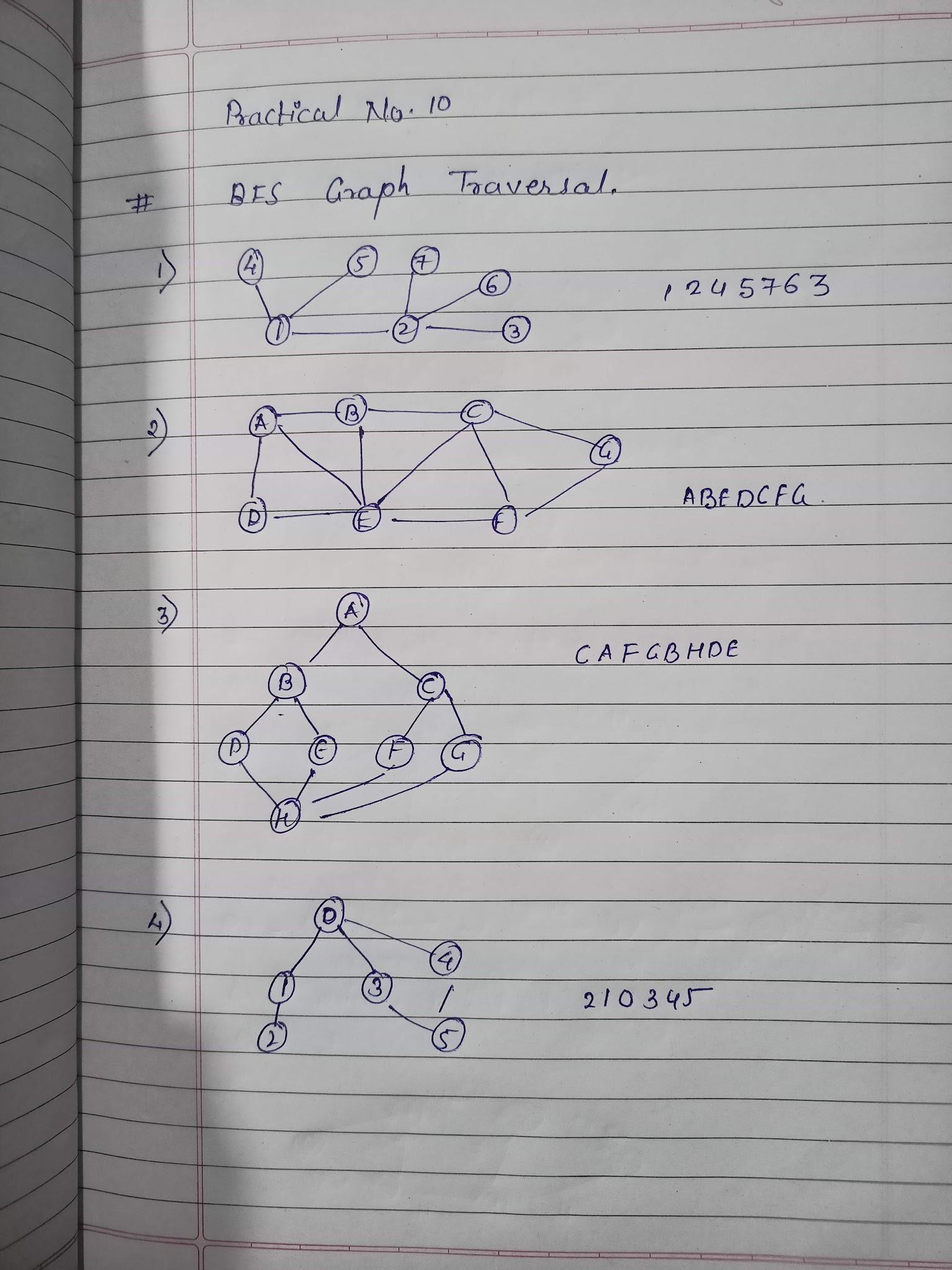
**Aim :** Solving problems on network flows using Ford - Fulkerson Labelling Algorithm.





**Practical no 10**

**Aim :** Solving problems on BFS Graph Traversal.

****

**Practical no 11**

**Aim** : Write a python program to demonstrate the working of set.

**Source code:**

#Python program to demonstrate the working# of

#Set in Python

#Creating two sets

set1=set()

set2=set()

#adding elements of set 1

for i in range (1,6):

set1.add(i)

#adding elements of set 2

for i in range(3,8):

set2.add(i)

print("set1=",set1)

print("set2=",set2)

print("\n")

#Union of set 1 and set 2

set3=set1 | set2 #set1.union(set2)

print("Union of Set1 & Set2: Set3=",set3)

#Intersection of set1 and set2

set4=set1 & set2 #set1.intersection(set2)

print("Intersection of Set1 & Set2: Set4=",set4)

print("\n")

#checking relation between set3 and set4

if set3>set4: #set3.issuperset(set4)

print("Set3 is superset of Set4")

elif set3<set4: #set3.issubset(set4)

print("Set3 is subset of Set4")

else: #set3==set4

print("Set3 is same as Set4")

#displaying relation between set4 and set3

if set4<set3: #set4.issubset(set3)

print("Set4 is subset of Set3")

print("\n")

#difference between set3 and set4

set5=set3-set4

print("Element in Set3 and not in Set4: Set5=",set5)

print("\n")

#checkv if set4 and set5 are disjoints sets

if set4.isdisjoint(set5):

print("Set4 and Set5 have nothing in common\n")

#Removing all the values of set5

set5.clear()

print("after applying clear on sets Set5:")

print("Set5=",set5)

**Output:**

