PRACTICAL1: MATRIX MULTIPLICATION

INPUT:

```
print("Enter order of 1st matrix:")
a,b = list(map(int,input().split()))

print("Enter Row wise values")


mat1 = []

for i in range(a) :
    print("Enter row",i,"value:")

    row = list(map(int,input().split()))
    mat1.append(row)

print("Enter order of 2nd matrix:")

d,c = list(map(int,input().split()))

print("Enter Row wise values")

mat2 = []

for j in range(d) :
    print("Enter row",j,"value:")


    row = list(map(int,input().split()))
    mat2.append(row)

print("Matrix 1:",mat1)
print("Matrix 2:",mat2)

resultant = []

for i in range(a):
    row = []
    for j in range(c):
        row.append(0)

    resultant.append(row)
```

```
print("Matrix Multiplication: ")
for i in range(a):
    for j in range(c):
        for k in range(b) :
            resultant[i][j] += mat1[i][k] * mat2[k][j]


for row in resultant:
    print(row)
```

OUTPUT:

```
Enter order of 1st matrix:
Enter row 0 value:
1 3 3
Enter row 1 value:
2 1 9
Enter row 2 value:
4 1 9
Matrix 1: [[1, 2, 3], [1, 2, 3], [1, 2, 3]]
Matrix 2: [[1, 3, 3], [2, 1, 9], [4, 1, 9]]
Matrix Multiplication:
[17, 8, 48]
[17, 8, 48]
[17, 8, 48]
```

PRACTICAL 2: QUICK SORT

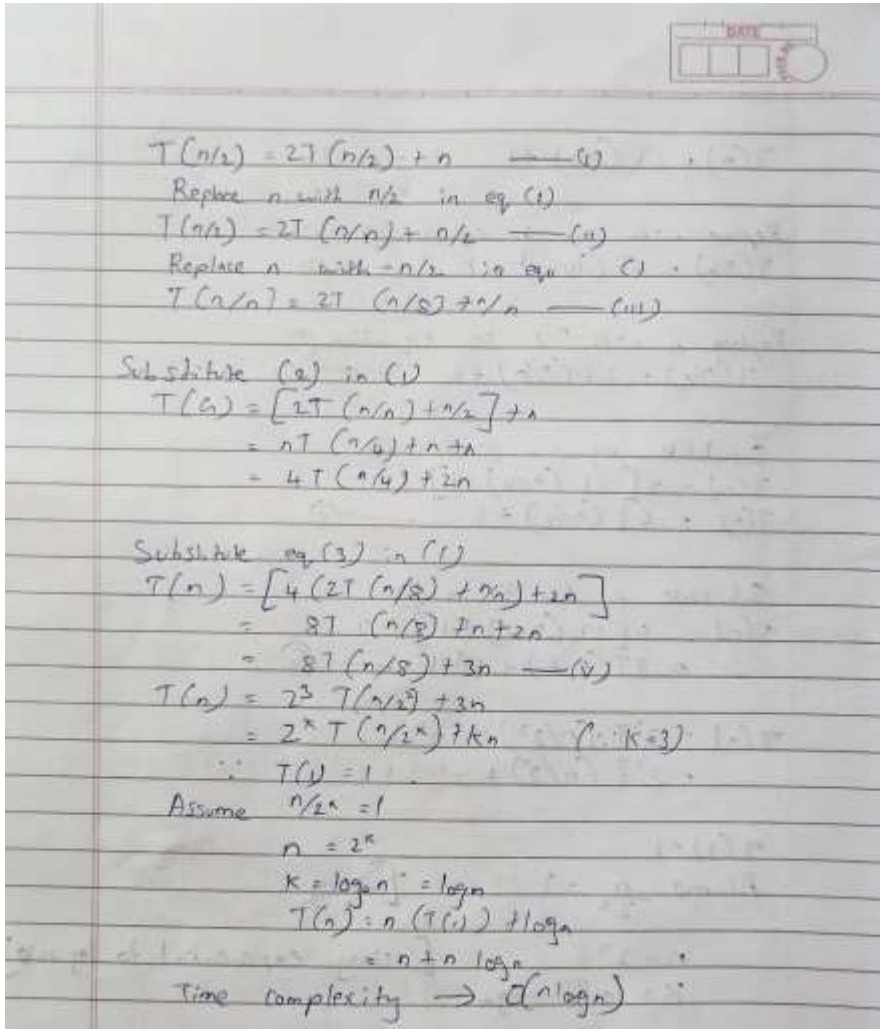
INPUT:

```
def partition(array, low, high):


    pivot = array[high]


    i = low - 1

    for j in range(low, high):
        if array[j] <= pivot:

            i = i + 1


            (array[i], array[j]) = (array[j], array[i])


    (array[i + 1], array[high]) = (array[high], array[i + 1])

       return i + 1



def quickSort(array, low, high):
    if low < high:

        pi = partition(array, low, high)
        quickSort(array, low, pi - 1)
         quickSort(array, pi + 1, high)


data = [2, 1, 9, 4, 5, 9, -2]
print("Unsorted Array")
print(data)

size = len(data)

quickSort(data, 0, size - 1)

print('Sorted Array in Ascending Order:')
print(data)
```

OUTPUT:

```
Unsorted Array
[2, 1, 9, 4, 5, 9, -2]
Sorted Array in Ascending Order:
[-2, 1, 2, 4, 5, 9, 9]
```

Time Complexity:



PRACTICAL 3: MERGE SORT

INPUT:

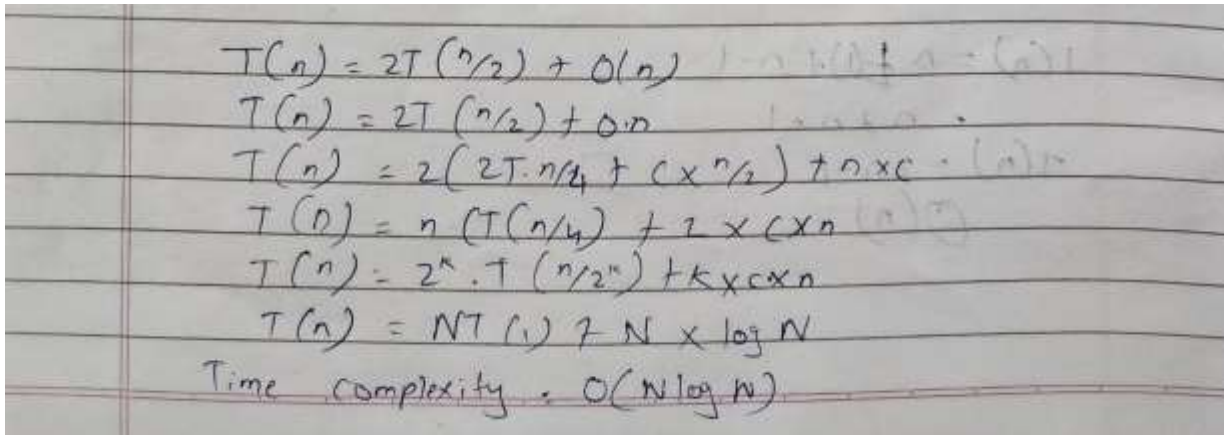
```
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m


    L = [0] * (n1)
    R = [0] * (n2)

    for i in range(0, n1):
        L[i] = arr[l + i]

    for j in range(0, n2):
        R[j] = arr[m + 1 + j]


    i = 0
    j = 0
    k = l
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1


    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1

    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1
```

```
def mergeSort(arr, l, r):
    if l < r:


        m = l+(r-l)//2


        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)

arr = [4, 1, 5, 9, 2, 7]
n = len(arr)
print("Given array is")
for i in range(n):
    print("%d" % arr[i],end=" ")

mergeSort(arr, 0, n-1)
print("\n\nSorted array is")
for i in range(n):
    print("%d" % arr[i],end=" ")
```

OUTPUT:

```
Given array is
4 1 5 9 2 7

Sorted array is
1 2 4 5 7 9
```

Time Complexity:



PRACTICAL 4: LINEAR SEARCH

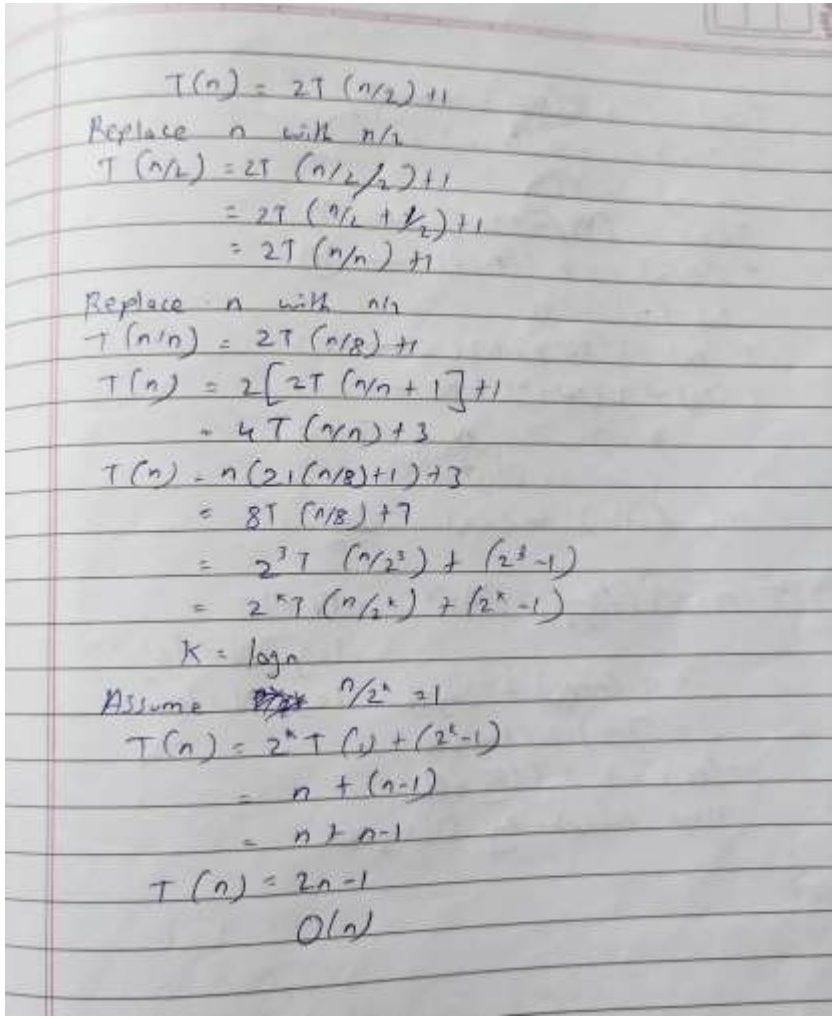
INPUT:

```
def linearSearch(array, n, x):


    for i in range(0, n):
        if (array[i] == x):
            return i
    return -1

array = [2, 4, 0, 1, 9]
x = 1
n = len(array)
result = linearSearch(array, n, x)
if(result == -1):
    print("Element not found")
else:
    print("Element found at index: ", result)
```

OUTPUT:

```
Element found at index:  3
```

Time Complexity:

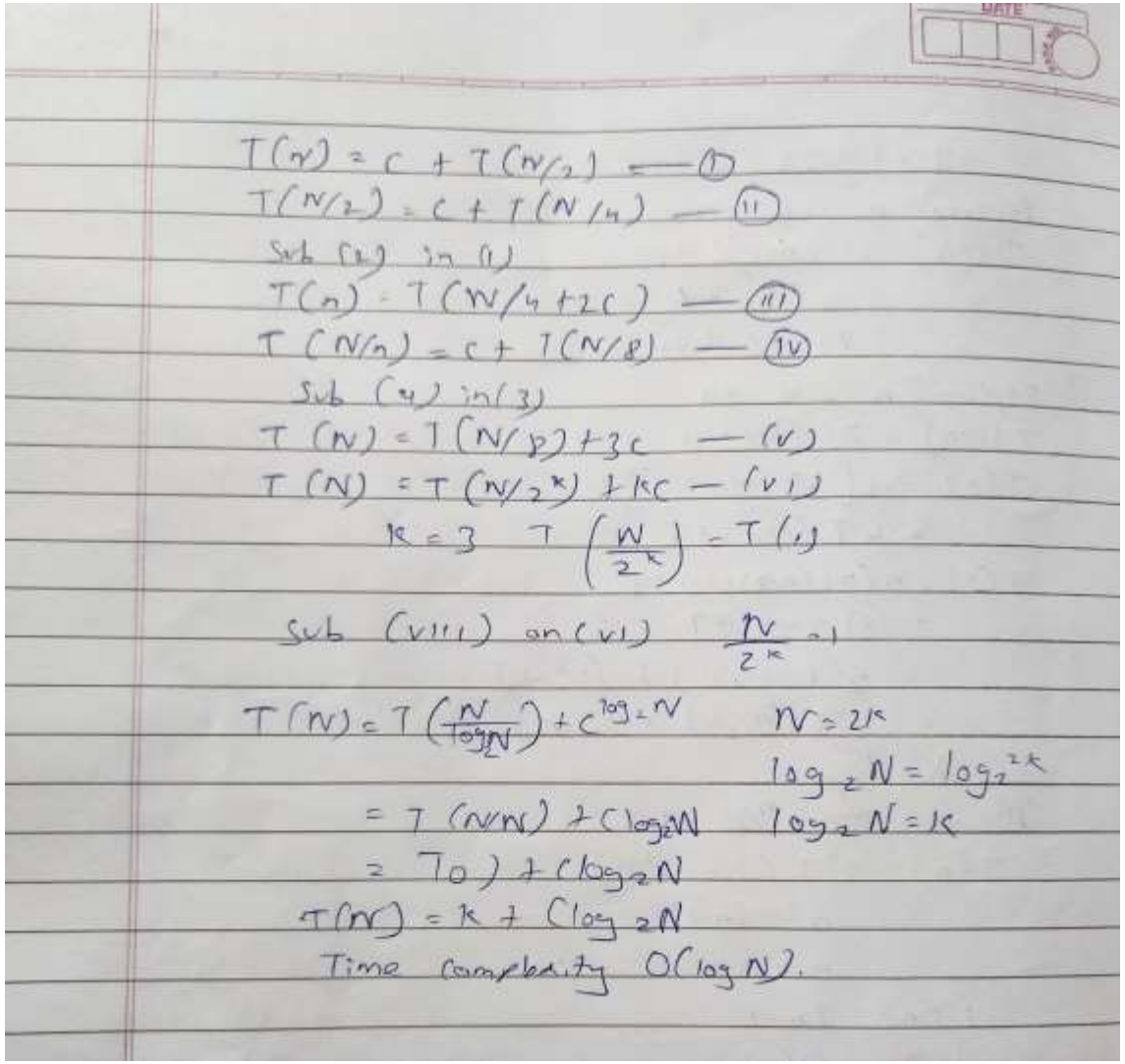


PRACTICAL 5

INPUT:

```
def binary_search(arr, low, high, x):


    if high >= low:

        mid = (high + low) // 2


        if arr[mid] == x:
            return mid


        elif arr[mid] > x:
            return binary_search(arr, low, mid - 1, x)


        else:
            return binary_search(arr, mid + 1, high, x)

    else:

        return -1


arr = [ 2, 3, 4, 10, 40 ]
x = 10


result = binary_search(arr, 0, len(arr)-1, x)

if result != -1:
    print("Element is present at index", str(result))
else:
    print("Element is not present in array")
```

Time Complexity:



OUTPUT:

```
Element found at index:  3
```

PRACTIACAL NO:-06

Aim:-Write a program to implement insertion operation on Binary Search tree and discuss the time complexity of the algorithm. Code:- class Node: def __init__(self,index):

```
    self.left=None
self.right=None
self.val=index

def insert(root,newnode):  if
root is None:
root=newnode  else:  if
root.val<=newnode.val:
if root.right is None:
root.right=newnode
else:
        insert(root.right,newnode)
elif root.val==newnode.val:
    print("Already existing node ",newnode.val)
else:
    if root.left is None:
root.left=newnode
else:
        insert(root.left,newnode)

def inorder(root):  if
root:  if
root.val==None:
```

```
print(end="")
else:
        inorder(root.left)
print(root.val)
inorder(root.right)

def preorder(root):
if root:        if
root.val==None:
print(end="")
else:
        print(root.val)
preorder(root.left)
preorder(root.right)

def postorder(root):
if root:
      if root.val==None:
print(end="")        else:
        print(root.val)
postorder(root.left)
postorder(root.right)
root=Node(100)
insert(root,Node(60))
insert(root,Node(50))
insert(root,Node(90))
```

```
insert(root,Node(40))
insert(root,Node(53))
insert(root, Node(95))
insert(root, Node(75))


print("inorder Traversal:")
inorder(root)
print("preorder Traversal:")
preorder(root)
print("postorder Traversal:")
postorder(root)

print("\n")
```

#Time Complexity:-

The time complexity for inserting node **depends on the height of the tree h** , so O(n)

Output:-

```
inorder Traversal:
40
50
53
60
75
90
95
100
preorder Traversal:
100
60
50
40
53
90
75
95
postorder Traversal:
100
60
50
40
53
90
75
95
```

PRACTIACAL NO:-07

Aim:- Write a python program to delete node from the binary tree from a given data and discuss the time complexity. Code:- class Node: def __init__(self,index):

```
    self.left=None
self.right=None
self.val=index

def insert(root,newnode):  if
root is None:
root=newnode  else:  if
root.val<=newnode.val:
if root.right is None:
root.right=newnode
else:
        insert(root.right,newnode)
elif root.val==newnode.val:
      print("Already existing node ",newnode.val)
else:
      if root.left is None:
root.left=newnode
else:
        insert(root.left,newnode)

def inorder(root):  if
root:  if
root.val==None:
```

```
print(end="")
else:
        inorder(root.left)
print(root.val)
inorder(root.right)

def preorder(root):
if root:        if
root.val==None:
print(end="")
else:
        print(root.val)
preorder(root.left)
preorder(root.right)

def postorder(root):
if root:
      if root.val==None:
print(end="")        else:
        print(root.val)
postorder(root.left)
postorder(root.right) def
delete(root,node1):     if root
is None:        print("Empty
tree")     elif root.val<node1:
delete(root.right,node1)
```

```
elif root.val>node1:
delete(root.left,node1)
else:
    root.val=None

root=Node(100)
insert(root,Node(60))
insert(root,Node(50))
insert(root,Node(90))
insert(root,Node(40))
insert(root,Node(53))
insert(root, Node(95))
insert(root, Node(75))

print("Before Deletion Operation: \n")
print("inorder Traversal:")
inorder(root) print("preorder
Traversal:") preorder(root)
print("postorder Traversal:")
postorder(root)

print("\n")

delete(root,40) delete(root,90)
print("After Deletion Operation: \n")
print("inorder Traversal:")
```

inorder(root) print("preorder

Traversal:") preorder(root)

print("postorder Traversal:")

postorder(root)

#Time Complexity:-

The time complexity for deleting node , so O(n)

Output:-

```
>>>
Before Deletion Operation:
inorder Traversal:
40
50
53
60
75
90
95
100
preorder Traversal:
100
60
50
40
53
90
75
95
postorder Traversal:
100
60
50
40
53
90
75
95
After Deletion Operation:
inorder Traversal:
50
53
60
100
preorder Traversal:
100
60
50
53
postorder Traversal:
100
60
50
53
>>>
```

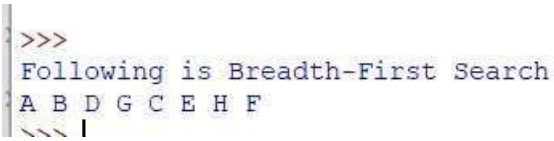
PRACTIACAL NO:-08

Aim:- Write a python program to implement Breadth First Traversal of graph.

Code:-
```
def bfs(visited, graph, node):
    visited.append(node)
queue.append(node)

    while queue:
        m=queue.pop(0)
print(m, end=" ")

        for neighbour in graph[m]:
if neighbour not in visited:
visited.append(neighbour)
queue.append(neighbour)

graph = {
        'A': ['B', 'D', 'G'],
        'B': ['A', 'C', 'D'],
        'C': ['B', 'E', 'F'],
        'D': ['A', 'B', 'E'],
        'E': ['D', 'C', 'F', 'G'],
        'F': ['C', 'E', 'H'],
        'G': ['A', 'E', 'H'],
        'H': ['G', 'F'],
        }
visited=[] queue=[]
```

print("Following is Breadth-First Search") bfs(visited,graph,'A')

Output:-



PRACTIACAL NO:-09

Aim:- Write a python program to implement Depth First Traversal of graph.

Code:-

```
# Using a Python dictionary to act as an adjacency list graph

= {

  'A': ['B', 'D', 'G'],

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

      'C': ['B', 'E', 'F'],

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

      'E': ['D', 'C', 'F', 'G'],

      'F': ['C', 'E', 'H'],

      'G': ['A', 'E', 'H'],

      'H': ['G', 'F']}



visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node): #function for dfs

if node not in visited: print (node,end=" ")

visited.add(node) for neighbour in
```

```
graph[node]:            dfs(visited, graph,
neighbour)


# Driver Code
print("Following is the Depth-First Search") dfs(visited,
graph, 'A')
```

Output:-

```
>>>
Following is the Depth-First Search
A B C E D F H G
>>>
```

PRACTIACAL NO:-10

Aim:- Write python program for checking whether a given graph g has a simple path from source s to destination d Code:-

```
graph={'A':['B','C'],
    'B':['C','D'],
     'C':['D'],
     'D':['C'],
     'E':['F'],
      'F':['C']}
def find_all_paths(graph,start,end,path=[]):
   path=path+[start]#source vertex included in path
if start==end:        return[path]     if start not in
graph:
       print("start vertex is not present in the graph")
return None     paths=[]
```

```
    for node in graph[start]:#iterating through graph
if node not in path:
            newpaths=find_all_paths(graph,node,end,path)
for newpath in newpaths:


                paths.append(newpath)
return paths


find_all_paths(graph,'A','D')
```

Output:-

```
>>>
>>> find_all_paths(graph,'A','D')
[['A', 'B', 'C', 'D'], ['A', 'B', 'D'], ['A', 'C', 'D']]
>>>
```

PRACTIACAL NO:-11

Aim:- Write a python program to implement selection sort algorithm and discuss the time complexity Code:- a=list()

```
n=int(input("Enter the number of elements in the list:-"))
print("Enter numbers in array") for i in range(n):
num=input()
    a.append(int(num))


print(a)


for i in range(len(a)):
    min_index = i     for j in
range(i+1,len(a)):        if
a[min_index]>a[j]:
```

```
        min_index=j
    a[min_index],a[i]=a[i],a[min_index]
print("Iteration : ",(i+1))     print(a)


print("Smallest element is : ",a[0])
print("Largest element is : ",a[len(a)-1])
```

#Time Complexity:-

The time complexity for selection sort algorithm is **$O(n^2)$ in all three cases**

Output:-

```
>>>
Enter the number of elements in the list:-5
Enter numbers in array
2
3
8
9
7
[2, 3, 8, 9, 7]
Iteration :  1
[2, 3, 8, 9, 7]
Iteration :  2
[2, 3, 8, 9, 7]
Iteration :  3
[2, 3, 7, 9, 8]
Iteration :  4
[2, 3, 7, 8, 9]
Iteration :  5
[2, 3, 7, 8, 9]
Smallest element is :  2
Largest element is :  9
>>>
```

PRACTIACAL NO:-12

Aim:- Using Tournament method find the second largest number in the given list.

Code:-

```
groups=[]
def largest(list1):
    global groups

    if len(list1)==1:
        #print("*",list1[0])
    return list1[0]    else:
        left=largest(list1[:len(list1)//2])
        #print("left",left)
right=largest(list1[len(list1)//2:])
#print("right",right)
groups.append((left,right))
#print("max",max (left,right))        return
max (left,right)

l1=largest([103,10,9,50,60,30])
print("First Largest is:-",l1) s=[]
for item in groups:        if l1
in item:
        s.append(min(item))
print("Second largest is:-",max(s))
```

Output:-

```
>>>
First Largest is:- 103
Second largest is:- 60
>>>
```