INDEX

12	largest number in the given list.	**33**	

PRACTIACAL NO:-01

Aim:-Write a program to implement matrix multiplication and discuss the time complexity of the algorithm. Code:-

```
m=int(input("Enter number of rows matrix1"))

n=int(input("Enter number of coln matrix1")) mat1=[]

#creating no of rows for

i in range (0,m):

mat1.append([])


#for above     #time complexity :->>> (c1)*n


print(mat1)

#creating no of columns for every rows

for i in range(0,m):     for j in

range(0,n):        mat1[i].append(j)

mat1[i][j]=0

      print("Entry in row:",i+1,"Column:",j+1)

mat1[i][j]=int(input())        print(mat1[i][j])

print(mat1)


#for above #time complexity :->>> (c2)*n^2
```

```
p=int(input("Enter number of rows matrix2"))

q=int(input("Enter number of coln matrix12"))

mat2=[] #creating no of rows for i in range

(0,p):    mat2.append([]) print(mat2)
```

#for above #time complexity :->>> (c3)*n

```
#creating no of columns for every rows

for i in range(0,p):    for j in range(0,q):

mat2[i].append(j)       mat2[i][j]=0

        print("Entry in row:",i+1,"Column:",j+1)

mat2[i][j]=int(input())       print(mat1[i][j])

print(mat2)
```

#for above #time complexity :->>> (c4)*n^2

```
mat3=[] for i in

range (0,m):

mat3.append([])
```

#for above #time complexity :->>> (c5)*n

```
for i in range(0,m):

for j in range(0,q):
```

```
mat3[i].append(j)

mat3[i][j]=0


    #for above #time complexity :->>> (c6)*n^2


print("multiplication result") if

n!=p:

   print("Multiplication not possible") else:

   for p in range (len(mat1)):

for q in range (len(mat2[0])):

for r in range (len(mat2)):

            mat3[p][q]+=mat1[p][r]*mat2[r][q]
print(mat3)


#for above #time complexity :->>> (c7)*n^3


#T(n)=(c7)*n^3+(c6)*n^2+(c5)*n+(c4)*n^2+(c3)*n+(c2)*n^2+(c1)*n #Highest
power is n^3 then big-O (n^3)
```

Output:-

```
Enter number of rows matrix12
Enter number of coln matrix12
[[], []]
Entry in row: 1 Column: 1
2
2
Entry in row: 1 Column: 2
2
2
Entry in row: 2 Column: 1
2
2
Entry in row: 2 Column: 2
2
2
[[2, 2], [2, 2]]
Enter number of rows matrix22
Enter number of coln matrix121
[[], []]
Entry in row: 1 Column: 1
3
2
Entry in row: 2 Column: 1
3
2
[[3], [3]]
multiplication result
[[12], [12]]
```

PRACTIACAL NO:-02

Aim:-Write a program to implement quick sort algorithm and discuss the time complexity of the algorithm.

Code:-

```
# Creating a quicksort function   def

quicksort (testlist,start,end):   #T(n)   if

start<end:
```

```
        pivot=partition(testlist,start,end) #n
quicksort(testlist,start,pivot-1) #T(n/2)
quicksort(testlist,pivot+1,end) #T(n/2)
return testlist

def partition(testlist,start,end):
pivot=testlist[end]
i=start-1       for j in range
(start,end):            if
testlist[j]<=pivot:
                        i+=1
                        testlist[i],testlist[j]=testlist[j],testlist[i]
testlist[i+1],testlist[end]=testlist[end],testlist[i+1]   return (i+1)
list1=[9,-3,5,2,6,8,-6,1,3] print("list
before sorting=",list1) print("After
sorting")
l2=quicksort(list1,0,len(list1)-1)
print(l2)
```

#Time Complexity #T(n)=2T(n/2)+n

$$T(n) = 2T(n/2) + n \quad \text{— (I)}$$

Replace n with n/2 in eq. (I)

$$T(n/2) = 2T(n/4) + n/2 \quad \text{— (II)}$$

Replace n with n/2 in eq. (II)

$$T(n/4) = 2T(n/8) + \frac{n}{4} \quad \text{— (III)}$$

substitute eq. (2) in (1)

$$T(n) = 2\left[2T(n/4) + \frac{n}{2}\right] + n$$

$$= 4T\left(\frac{n}{4}\right) + n + n$$

$$= 4T\left(\frac{n}{4}\right) + 2n \quad \text{— (IV)}$$

substitute eq. (3) in (1)

$$T(n) = \left[4\left[2T(n/8) + \frac{n}{4}\right]\right] + 2n$$

$$= 8T\left(\frac{n}{8}\right) + n + 2n$$

$$= 8T(n/8) + 3n \quad \text{— (V)}$$

$$T(n) = 2^3 T(n/2^3) + 3n$$

$$= 2^k T\left(\frac{n}{2^k}\right) + kn \quad \ldots (k=3)$$

$\because T(1) = 1$

Assume $\frac{n}{2^k} = 1$

$$n = 2^k$$

$$k = \log_2 n = \log n$$

$$T(n) = n(T(1)) + \log n \cdot n$$

$$= n + n \cdot \log n$$

time complexity $\Rightarrow O(n \log n)$

Output:-

```
>>>
list before sorting= [9, -3, 5, 2, 6, 8, -6, 1, 3]
After sorting
[-6, -3, 1, 2, 3, 5, 6, 8, 9]
```

PRACTIACAL NO:-03

Aim:- Write a program to implement merge sort algorithm and discuss the time complexity of the algorithm. Code:- def mergesort(mylist):

print("Dividing:",mylist)   if(len(mylist)>1):

    mid=len(mylist)//2

    Leftlist=mylist[:mid]

Rightlist=mylist[mid:]

```
mergesort(Leftlist)
mergesort(Rightlist)
    i=0        j=0        k=0
while(i<len(Leftlist) and (j<len(Rightlist))):
        if Leftlist[i]<=Rightlist[j]:
mylist[k]=Leftlist[i]
i=i+1            else:
            mylist[k]=Rightlist[j]
j=j+1            k=k+1        while
i<len(Leftlist):
mylist[k]=Leftlist[i]
i=i+1            k=k+1
    while j<len(Rightlist):
mylist[k]=Rightlist[j]
j=j+1            k=k+1
    print("merging:",mylist)
return mylist
num=int(input("How many number you want in list:"))
list1=[int(input())for x in range(num)] mergesort(list1)
print("Sorted list: ",list1)
```

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$$T(n) = 2T\left(\frac{n}{2}\right) + C.n$$

$$T(n) = 2\left(2T\left(\frac{n}{4}\right) + c \times \frac{n}{2}\right) + n \times c$$

$$T(N) = 4\left(T\left(\frac{n}{4}\right)\right) + 2 \times c \times n$$

$$T(n) = 2^K . T\left(\frac{n}{2^K}\right) + K \times c \times n$$

$$T(N) = N\, T(1) + N \times \log N$$

Time Complexity $= O(N \log N)$

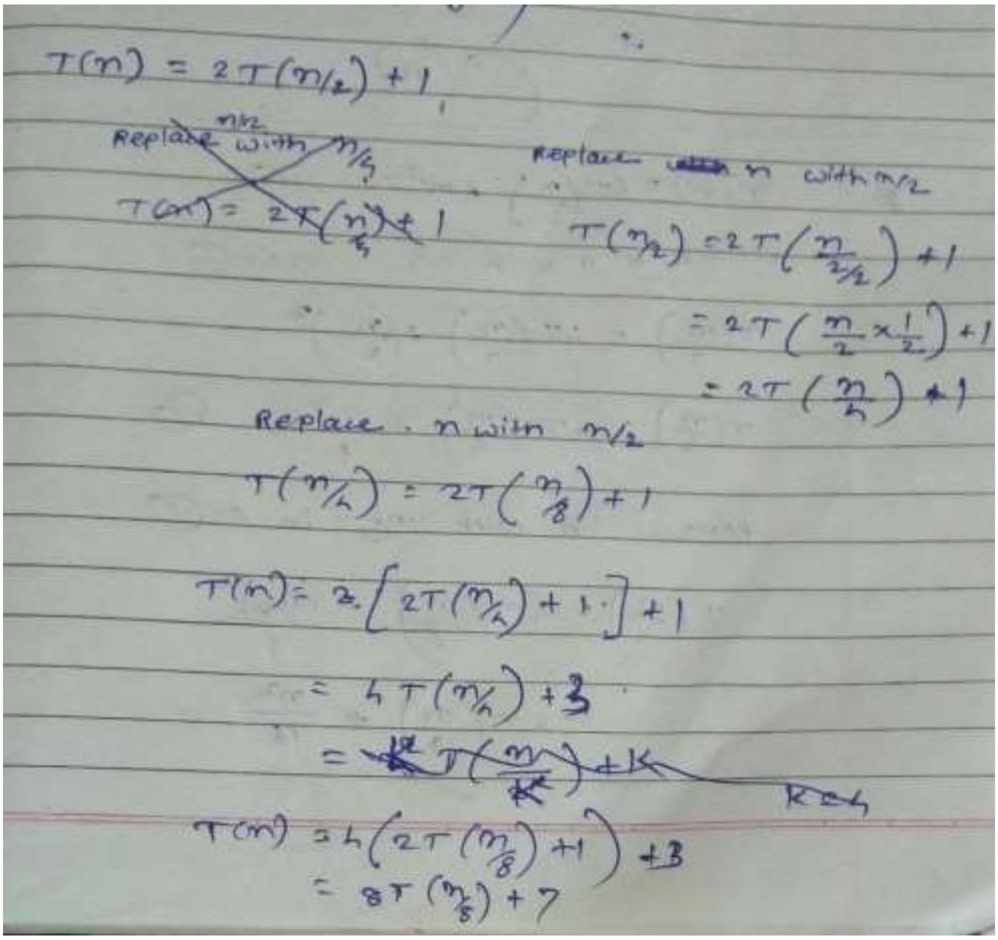
Output:-

```
>>>
How many number you want in list:5
1
2
8
9
5
Dividing: [1, 2, 8, 9, 5]
Dividing: [1, 2]
Dividing: [1]
Dividing: [2]
merging: [1, 2]
Dividing: [8, 9, 5]
Dividing: [8]
Dividing: [9, 5]
Dividing: [9]
Dividing: [5]
merging: [5, 9]
merging: [5, 8, 9]
merging: [1, 2, 5, 8, 9]
Sorted list:  [1, 2, 5, 8, 9]
```

PRACTIACAL NO:-04

Aim:-Write a program to implement Linear Search algorithm and discuss the time complexity of the algorithm. Code:- def linearsearch(mylist,n,x):

```
for i in range(0,n):                if (mylist[i]==x):                return i
return -1
mylist=[2,33,45,1,30,34,56] x=1
result=linearsearch(mylist,7,x)
if (result==-1):
   print("Element is not Present in list") else:
   print("Element is present and at index of",result[])
#Time Complexity #O(n)
```



$$= 2^3 T\left(\frac{n}{2^3}\right) + (2^3 - 1)$$

$$= 2^K T\left(\frac{n}{2^K}\right) + (2^K - 1) \ldots$$

$$K = \log n$$

Assume $\cancel{T(1)}$ $\frac{n}{2^K} = 1$

$$T(n) = 2^K T(1) + (2^K - 1)$$

$$= n \cancel{T} + (n-1)$$

$$= n + n - 1$$

$$T(n) = 2n - 1$$

$$O(n)$$

Output:-

```
>>>
Element is present and at index of 3
>>>
```

PRACTIACAL NO:-05

Aim:-Write a program to implement Binary Search algorithm and discuss the time complexity of the algorithm. Code:- 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=[2,3,6,11,13,8]
x=3 print(arr)
print("X=",x)
result=binary_search(arr,0,len(arr)-1,x) if(result!=1):
         print("Element is not Found") else:
      print("Element is prsent at a index ",result)
         #Time Complexity #T(n) = T(n/2) + c
#O(log n)
```

Binary search:-

$$T(N) = c + T(N/2) \quad \text{— (I)}$$

$$T(N/2) = c + T(N/4) \quad \text{— (II)}$$

substitute (II) in (I)

$$T(N) = T(N/4) + 2c \quad \text{— (III)}$$

$$T\left(\frac{N}{4}\right) = c + T(N/8) \quad \text{— (IV)}$$

substitute (IV) in (III)

$$T(N) = T(N/8) + 3c \quad \text{— (V)}$$

$$T(N) = T\left(\frac{N}{2^k}\right) + kc \quad \text{— (VI)}$$

$k = 3 \qquad T\left(\frac{N}{2^k}\right) = T(1)$

substitute (VII) in (VI)

$$\frac{N}{2^k} = 1$$

$$N = 2^k$$

$$\log_2 N = \log_2 2^k$$

$$\log_2 N = k \quad \text{(VII)}$$

$$T(N) = T\left(\frac{N}{2^{\log_2 N}}\right) + c\log_2 N$$

$$= T(N/N) + c\log_2 N$$

$$= T(1) + c\log_2 N$$

$$T(N) = k + c\log_2 N$$

Time complexity = $O(\log n)$

Output:-

```
>>>
[2, 3, 6, 11, 13, 8]
X= 3
Element is prsent at a index  1
>>>
```

```
>>>
[2, 3, 6, 11, 13, 8]
X= 5
Element is not Found
>>>
```

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:-

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

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
>>>
```