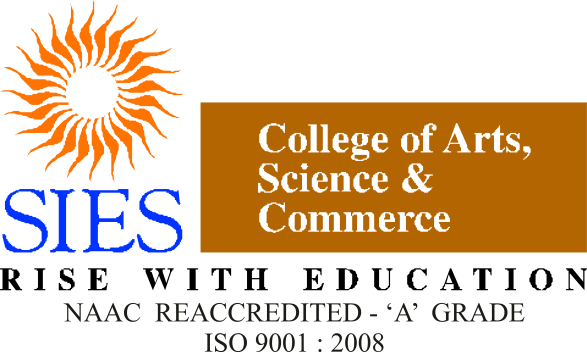
Name: - Chauhan Pankaj Yamunaprasad

Roll No.: - SCS2223008

Course: - SYBsc CS

Semester: - IV

Subject: - Fundamentals of Algorithm Journal



**S.I.E.S College of Arts, Science and Commerce**

**Sion(W), Mumbai – 400 022.**

**CERTIFICATE**

This is to certify that Mr. / ~~Miss~~. **CHAUHAN PANKAJ YAMUNAPRASAD**

Roll No **SCS2223008** Has successfully completed the necessary course of experiments in the subject of **Fundamentals Of Algorithm** during the academic year **2022 – 2023** complying with the requirements of **University of Mumbai**, for the course of **S.Y.BSc. Computer Science [Semester-IV]**

Prof. In-Charge

**Soni Yadav**

**(Fundamentals Of Algorithm)**

Examination Date:

Examiner’s Signature & Date:

HOD’s Signature & Date:

**Dr. Manoj Singh**

College Seal

And

 Date

INDEX

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Sr No. | Aim | Date |  |  |  | Pg No. |
| 1 | Write a python program to implement matrix multiplication by taking input from user and discuss time complexity of the algorithm | 20/12/22 |  |  |  | 3 |
| 2 | Write a python program to implement quick sort algorithm and discuss time complexity of the algorithm | 11/1/23 |  |  |  | 7 |
| 3 | Write a python program to implement merge sort algorithm and discuss time complexity of the algorithm | 11/1/23 |  |  |  | 10 |
| 4 | Write a python program to implement linear search algorithm and discuss time complexity of the algorithm | 16/1/23 |  |  |  | 13 |
| 5 | Write a python program to implement binary search algorithm and discuss time complexity of the algorithm | 29/1/23 |  |  |  | 15 |
| 6 | Write a python program to implement insertion operation in binary search tree and discuss time complexity | 9/2/23 |  |  |  | 18 |
| 7 | Write a python program to delete node from the binary search tree from a given data and discuss time complexity | 15/2/23 |  |  |  | 21 |
| 8 | Write a python program to implement Breadth First Traversal of graph | 18/2/23 |  |  |  | 23 |
| 9 | Write a python program to implement Depth First Traversal of graph | 18/2/23 |  |  |  | 24 |
| 10 | Write a python program for checking whether a given graph g has a simple path from source s to destination d | 22/2/23 |  |  |  | 25 |
| 11 | Write a python program to implement selection sort algorithm and discuss time complexity | 3/3/23 |  |  |  | 26 |
| 12 | Using Tournament method find the second largest number in the given list | 3/3/23 |  |  |  | 28 |

**Practical 1**

Aim :- Write a python program to implement matrix multiplication by taking input from user and discuss time complexity of the algorithm

Code :-

#Matrix 1

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

n = int(input("Enter number of coln matrix1:"))

mat1 =[]

for i in range (0,m):

mat1.append([])

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)

#matrix 2

p = int(input("Enter number or rows matrix 2: "))

q = int(input("Enter number or colns matrix 2: "))

mat2 = []

for i in range(0,p):

mat2.append([])

for i in range(0,p):

for j in range(0,p):

mat2[i].append(j)

mat2[i][j] = 0

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

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

print(mat2)

res = []

for i in range (0,m):

res.append([])

for i in range(0,m):

for j in range(0,q):

res[i].append(j)

res[i][j] = 0

#results

print("Multiplication Result: ")

if n!=p:

print("Multiplication is not possible")

else:

for p in range (len(mat1)):

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

for r in range (len(mat2)):

res[p][q] += mat1[p][r]\* mat2[r][q]

print(res)

OUTPUT: -

Enter number of rows matrix1:2

Enter number of coln matrix1:2

Entry in row: 1 column: 1

2

Entry in row: 1 column: 2

2

Entry in row: 2 column: 1

2

Entry in row: 2 column: 2

2

[[2, 2], [2, 2]]

Enter number or rows matrix 2: 2

Enter number or colns matrix 2: 2

Entry in row: 1 column: 1

2

Entry in row: 1 column: 2

2

Entry in row: 2 column: 1

2

Entry in row: 2 column: 2

2

[[2, 2], [2, 2]]

Multiplication Result:

[[8, 8], [8, 8]]

Time Complexity: -

Let c1 be the constant time taken by the statements outside the loops

Let c2 be the constant time taken by the statements inside the first for-loop which will be executed 'n' times.

Let c3 be the constant time taken by the statements inside the first nested for-loops which will be executed 'n2' times.

Let c4 be the constant time taken by the statements inside the second for-loop which will be executed 'n' times.

Let c5 be the constant time taken by the statements inside the second nested for-loops which will be executed 'n2' times.

Let c6 be the constant time taken by the statements inside the third for-loop which will be executed 'n' times.

Let c7 be the constant time taken by the statements inside the third nested for-loops which will be executed 'n2' times.

Let c8 be the constant time taken by the statements inside the last nested for loops which will be executed 'n3' times.

T(n) = c1 + n.c2 + n2.c3 + n.c4 + n2.c5 + n.c6 + n2.c7 + n3.c8

T(n) = O(n3)

The time complexity of Matrix Multiplication is O(n3).

**Practical 2**

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

Code :-

def quicksort(testlist,start,end):

if start<end:

pivot=partition(testlist,start,end)

quicksort(testlist,start,pivot-1)

quicksort(testlist,pivot+1,end)

return testlist

def partition(testlist,start,end):

pivot=testlist[end]

i=start-1

for j in range(start,end):

if testlist[j]<=pivot:

i=i+1

testlist[i],testlist[j]=testlist[j],testlist[i]

testlist[i+1],testlist[end]=testlist[end],testlist[i+1]

return(i+1)

testlist=[5,2,9,8,3,4,6]

print("The Sorted List is : ",quicksort(testlist,0,6))

OUTPUT :-

The Sorted List is : [2, 3, 4, 5, 6, 8, 9]

Time Complexity :-

Best Case :

Let the time taken by one array be T(n/2) [As we are dividing it into two parts]

Let ‘c’ be the constant time taken for traversing elements of array

Therefore T(n) = 2T(n/2) + n.c …..(1)

Replacing ‘n’ by ‘n/2’ in (1) we get,

T(n/2) = 2T(n/4) + (n.c)/2 …..(2)

Replacing ‘n’ by ‘n/2’ in (2) we get,

T(n/4) = 2T(n/8) + (n.c)/4 …..(3)

Substituting (2) in (1) we get,

T(n) = 2[2T(n/4) + (n.c)/2] + n.c

T(n) = 4T(n/4) + n.c + n.c

T(n) = 4T(n/4) + 2(n.c) …..(4)

Substituting (3) in (4) we get,

T(n) = 4[2T(n/8) + (n.c)/4] + 2(n.c)

T(n) = 8T(n/8) + n.c + 2(n.c)

T(n) = 8T(n/8) + 3(n.c)

T(n) = 23T(n/23) + 3(n.c) …..(5)

Replacing 3 by k in (5) we get,

T(n) = 2kT(n/2k) + k(n.c) …..(6)

Let us assume n/2k = 1 n = 2k , k = log n …...(7)

Substituting (7) in (6) we get,

T(n) = n.T(1) + log n(n.c)

T(n) = O(n.log n)

Therefore, the time complexity of Quick Sort Is O(n.log n)

**Practical 3**

Aim : Write a python program to implement merge sort algorithm and discuss time complexity of the algorithm

Code :-

def MergeSort(mylist):

if len(mylist) > 1:

mid=len(mylist) // 2

mid=int(mid)

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] < leftlist[j]:

mylist[k] = leftlist[i]

i+=1

else:

mylist[k] = rightlist[j]

j+=1

k+=1

while i<len(leftlist):

mylist[k] = leftlist[i]

i+=1

k+=1

while j<len(rightlist):

mylist[k] = rightlist[j]

j+=1

k+=1

return mylist

mylist = [12,23,57,8,9,1,2,0,24]

print("Unsorted List : ", mylist)

sorted\_list = MergeSort(mylist)

print("Sorted List : ", sorted\_list)

output: **-**

Unsorted List : [12, 23, 57, 8, 9, 1, 2, 0, 24]

Sorted List : [0, 1, 2, 8, 9, 12, 23, 24, 57]

**T**ime Complexity :-

Let T(n) be the time taken for 'n' inputs.

The time taken by the first while-loop is T(n/2) because it has an if-else statement in it out of which only one gets executed.

The time taken by the second and third while-loop is T(n/2) as only half of the inputs are considered. Out of the two while-loops, only one will be executed.

T(n) = 2T(n/2) + n .....(1)

Replacing n by n/2 in (1) we get,

T(n/2) = 2T(n/4) + n/2 .....(2)

Replacing n by n/2 in (2) we get,

T(n/4) = 2T(n/8) + n/4 .....(3)

Substituting (3) in (2) we get,

T(n/2) = 2[2T(n/8) + n/4] + n/2

T(n/2) = 4T(n/8) + 2xn/4 + n/2

T(n/2) = 4T(n/8) + n/2 + n/2

T(n/2) = 4T(n/8) + 2xn/2

T(n/2) = 4T(n/8) + n .....(4)

Substituting (4) in (1) we get,

T(n) = 2[4T(n/8) + n] + n

T(n) = 8T(n/8) + 2n + n

T(n) = 8T(n/8) + 3n

T(n) = 23T(n/23) + 3n .....(5)

Replacing 3 by k in (5) we get,

T(n) = 2kT(n/2k) + kn .....(6)

Let us assume n/2k =1, n=2k , k = log n

T(n) = n.T(1) + n.log n

T(n) = n + n.log n

T(n) = O(n.log n)

Therefore, the time complexity of Merge Sort Is O(n.log n)

**Practical 4**

Aim :- Write a python program to implement linear search algorithm and discuss time complexity of the algorithm

Code :-

def LinearSearch(list1, x):

if not list1:

print("The list is empty")

return

list1.sort()

print("The sorted array is", list1)

for i in range(len(list1)):

if list1[i] == x:

print("The element is available at index", i)

return

print("The searching element is not available in the array")

list1 = [12, 34, 1, 23, 2, 78, 90]

LinearSearch(list1, 1) # Best case

LinearSearch(list1, 23) # Average Case

LinearSearch(list1, 24) # Worst Case

LinearSearch(list1, 90) # Worst Case

Output :-

The sorted array is [1, 2, 12, 23, 34, 78, 90]

The element is available at index 0

The sorted array is [1, 2, 12, 23, 34, 78, 90]

The element is available at index 3

The sorted array is [1, 2, 12, 23, 34, 78, 90]

The searching element is not available in the array

The sorted array is [1, 2, 12, 23, 34, 78, 90]

The element is available at index 6

Time Complexity :-

Let ‘c’ be the time taken by the statements outside the loop

Let ‘T(n) be the time taken by the statements in the for-loop

Therefore, T(n) = T(n) + c

T(n) = O(n)

Therefore, the time complexity for worst case of Linear Search Algorithm is O(n).

The time complexity for best case of Linear Search Algorithm is O(1)

**Practical 5**

Aim :- Write a python program to implement binary search algorithm and discuss time complexity of the algorithm

Code :-

def BinarySearch(list1,x,start,end):

list1.sort()

if(end>start):

mid=(start+end)/2

mid=int(mid)

if(list1[mid]==x):

print("the element is available at the index ",mid)

elif(list1[mid]<x):

BinarySearch(list1,x,mid+1,end)

else:

BinarySearch(list1,x,start,mid)

else:

print("The element is not available in the list")

list1=[2,3,4,5,67,78,99,15,12]

e=len(list1)

BinarySearch(list1,15,0,e) #Best case

BinarySearch(list1,100,0,e) #Worst case

BinarySearch(list1,99,0,e) #Worst case

BinarySearch(list1,4,0,e) #Average case

Output :

the element is available at the index 5

The element is not available in the list

the element is available at the index 8

the element is available at the index 2

Time Complexity :-

Let the time taken by the BinarySearch function be T(n)

Let the time taken by the two recursive BinarySearch function be T(n/2).

Let ‘c’ be the time taken by all other statements.

Therefore, T(n) = T(n/2) + c …..(1)

Replacing ‘n’ by ‘n/2’ in (1) we get,

T(n/2) = T(n/4) + c …..(2)

Replacing ‘n’ by ‘n/2’ in (2) we get,

T(n/4) = T(n/8) + c …...(3)

Substituting (2) in (1) we get,

T(n) = T(n/4) + c + c

T(n) = T(n/4) +2c …..(4)

Substituting (3) in (4) we get,

T(n) = T(n/8) + c + 2c

T(n) = T(n/8) + 3c

T(n) = T(n/23) + 3c …..(5)

Replacing 3 by ‘k’ in (5) we get,

T(n) = T(n/2k) + kc …..(6)

Let us assume n/2k = 1 n = 2k, k = log n …..(7)

Substituting (7) in (6) we get,

T(n) = T(1) + (log n) .c

T(n) = O(log n)

Therefore, the time complexity for worst case of Binary Search Algorithm is O(log n).

The time complexity for best case of Binary Search Algorithm is O(1).

**Practical 6**

Aim :- Write a python program to implement insertion operation in binary search tree and discuss time complexity

Code :-

class Node:

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

self.value=index

self.right=None

self.left=None

def Insert(root, newnode):

if root is None:

root=newnode

else:

if root.value<newnode.value:

if root.right is None:

root.right=newnode

else:

Insert(root.right,newnode)

elif root.value>newnode.value:

if root.left is None:

root.left=newnode

else:

Insert(root.left,newnode)

else:

print("The node", newnode.value , "is already present")

def Inorder(root):

if root:

Inorder(root.left)

print(root.value)

Inorder(root.right)

def PreOrder(root):

if root:

print(root.value)

PreOrder(root.left)

PreOrder(root.right)

def PostOrder(root):

if root:

PostOrder(root.left)

PostOrder(root.right)

print(root.value)

r = Node(12)

Insert(r,Node(13))

Insert(r,Node(15))

Insert(r,Node(17))

Insert(r,Node(11))

Insert(r,Node(10))

Insert(r,Node(19))

Insert(r,Node(20))

Insert(r,Node(1))

Insert(r,Node(2))

Insert(r,Node(27))

print("Inorder Traversal")

Inorder(r)

print("PostOrder Traversal")

PostOrder(r)

print("PreOrder Traversal")

PreOrder(r)

Output :-

Inorder Traversal

1

2

10

11

12

13

15

17

19

20

27

PostOrder Traversal

2

1

10

11

27

20

19

17

15

13

12

PreOrder Traversal

12

11

10

1

2

13

15

17

19

20

27

Time Complexity: -

In worst case, the binary search tree is a skewed binary search tree (either left or right) and we must travel from the root to the deepest leaf node. In that case, the height of the binary search tree becomes n Thus, in worst case, the time complexity of binary search tree for insertion operation becomes O(n), as for inserting a new element in a skewed Binary search tree, we must need to traverse through all the existing nodes first.

**Practical 7**

Aim :- Write a python program to delete node from the binary search tree from a given data and discuss time complexity

Code :-

class Node:

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

self.index = index

self.left = None

self.right = None

def Insert(node, index):

if node is None:

return Node(index)

if index < node.index:

node.left = Insert(node.left, index)

else:

node.right = Insert(node.right, index)

return node

def searchminnode(node):

current = node

while current.left is not None:

current = current.left

return current

def Inorder(root):

if root is not None:

Inorder(root.left)

print(root.index)

Inorder(root.right)

def deletenode(root, index):

if root is None:

return root

if index < root.index:

root.left = deletenode(root.left, index)

elif index > root.index:

root.right = deletenode(root.right, index)

else:

if root.left is None:

temp = root.right

root = None

return temp

elif root.right is None:

temp = root.left

root = None

return temp

temp = searchminnode(root.right)

root.index = temp.index

root.right = deletenode(root.right, temp.index)

return root

root = None

root = Insert(root, 100)

root = Insert(root, 20)

root = Insert(root, 10)

root = Insert(root, 50)

root = Insert(root, 120)

root = Insert(root, 182)

root = Insert(root, 118)

root = Insert(root, 90)

root = Insert(root, 180)

print("Inorder traversal of a tree is : ")

Inorder(root)

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("After deleting node 120")

root = deletenode(root, 120)

Inorder(root)

Output :-

Inorder traversal of a tree is :

10

20

50

90

100

118

120

180

182

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

After deleting node 120

10

20

50

90

100

118

180

182

Time Complexity :-

In worst case, the binary search tree is a skewed binary search tree (either left or right) and we must travel from the root to the deepest leaf node. In that case, the height of the binary search tree becomes n Thus, in worst case, the time complexity of binary search tree for deletion operation becomes O(n).

**Practical 8**

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

Code :-

from queue import Queue

graph = {'A': ['B', 'D'],

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

'C': ['B'],

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

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

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

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

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

#Initialization of required variables

visited={}

level={}

parent={}

bfs\_traversal\_output=[]

queue=Queue()

#Assigning default values

for node in graph.keys():

visited[node]=False

parent [node]=None

level [node ]=-1

# Taking source and assigning its default values

s=input("Enter the source ")

if s in graph:

visited[s]=True

level[s]=0

queue.put(s)

else:

print("The source is not available in the graph")

#Finding bfs

while not queue.empty(): #To check wether the queue is not empty

u=queue.get() #Storing the source in the variable

bfs\_traversal\_output.append(u)#storing the source

for v in graph[u]:#For the child of source

if not visited[v]:#If they are not visited

visited[v]=True #Visit Them

parent[v]=u #Parent is assigned

queue.put(v) #stored in the queue

print(bfs\_traversal\_output)

Output :-

Enter the source A

['A', 'B', 'D', 'C', 'E', 'F', 'G', 'H']

**Practical 9**

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

Code :-

graph={'A': ['B', 'H', 'G'],

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

'C': ['B'],

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

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

'F': ['G', 'I'],

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

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

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

def DFS(node, graph):

visited = set()

if node not in graph:

print("Node is not present in the graph")

return

stack = []

stack.append(node)

while stack:

current=stack.pop()

if current not in visited:

print(current)

visited.add(current)

for i in graph[current]:

stack.append(i)

print("Depth First Traversal of Graph with respect to A is : ")

DFS("A", graph)

Output :-

Depth First Traversal of Graph with respect to A is :

A

G

H

I

F

E

D

B

C

**Practical 10**

Aim :- Write a 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]

print (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]:

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 :-

['A']

['A', 'B']

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

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

['A', 'B', 'D']

['A', 'C']

['A', 'C', 'D']

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

**Practical 11**

Aim :- Write a python program to implement selection sort algorithm and discuss 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("Original array: ", a)

for i in range(len(a)):

min\_ind = i

for j in range(i+1, len(a)):

if a[min\_ind] > a[j]:

min\_ind = j

a[min\_ind], a[i] = a[i], a[min\_ind]

print("Iteration ", i+1)

print(a)

print("Smallest element is ", a[0])

print("Largest element is ", a[-1])

Output :-

Enter the number of elements in the list: 6

Enter numbers in array

21

23

34

12

12

0

Original array: [21, 23, 34, 12, 12, 0]

Iteration 1

[0, 23, 34, 12, 12, 21]

Iteration 2

[0, 12, 34, 23, 12, 21]

Iteration 3

[0, 12, 12, 23, 34, 21]

Iteration 4

[0, 12, 12, 21, 34, 23]

Iteration 5

[0, 12, 12, 21, 23, 34]

Iteration 6

[0, 12, 12, 21, 23, 34]

Smallest element is 0

Largest element is 34

Time Complexity :-

Let c1 be the time taken by the statement outside the loops.

Let c2 be the time taken by the statements in the first for loop which will be executed ‘n’ times.

Let c3 be the time taken by the statements in the second for loop which will be executed ‘n2’ times.

Let T(n) be the time taken by the Selection Sort Algorithm.

T(n) = c1 + n.c2 + n2.c3

T(n) = O(n2)

Therefore, the Time Complexity of Selection Sort Algorithm is O(n2).

**Practical 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:

return list1[0]

else:

left = largest (list1[:len(list1)//2])

right=largest (list1[len(list1)//2:])

groups.append((left, right))

print("Groups:",groups)

print("Maximum:",max(left, right))

return max(left, right)

l1 = largest([3,10,9,50,60,30])

s=[]

for item in groups:

print("Item : ",item)

if l1 in item:

print("largest Element: ",l1)

s.append(min(item))

print("Minimum among group-item: ",s)

print("Second Largest number is :" ,max(s))

Output :-

Groups: [(10, 9)]

Maximum: 10

Groups: [(10, 9), (3, 10)]

Maximum: 10

Groups: [(10, 9), (3, 10), (60, 30)]

Maximum: 60

Groups: [(10, 9), (3, 10), (60, 30), (50, 60)]

Maximum: 60

Groups: [(10, 9), (3, 10), (60, 30), (50, 60), (10, 60)]

Maximum: 60

Item : (10, 9)

Item : (3, 10)

Item : (60, 30)

largest Element: 60

Minimum among group-item: [30]

Item : (50, 60)

largest Element: 60

Minimum among group-item: [30, 50]

Item : (10, 60)

largest Element: 60

Minimum among group-item: [30, 50, 10]

Second Largest number is : 50