Lab # 1

Q # 01 . Discuss the characteristics of Algorithms.

Characteristics of Algorithm:

1. Input: Input is the data provided by the user that transforms into output after computations. An algorithm should have 0 or more well defined inputs.

2. Output: Output is the result from the computation of the input provided by the user. An algorithm must have 1 or more well defined outputs and it should match the desired output.

3. Definiteness: An algorithm should specify each step and the order the steps must be taken in the process. An algorithm must clearly define each step and it should be unambiguous.

4. Effectiveness: Effectiveness means that the algorithm should not contain any unnecessary and ambiguous steps, and it should produce the desired output with the available resources.

5. Finiteness: It means that the algorithm must stop immediately either if you get the desired output or you got an error for solution not possible.

6. Independent: An algorithm should be independent of any programming language. It should run in all the programming languages and produce the same desired output.

Q # 02 . How do you differentiate linear and non- linear data structures? Also give some examples of both data structures.

Linear Data Structures

1. Elements are arranged linearly means that each element is inserted adjacent to each other and can be accessed in the same manner.

2. Elements are based on a single level.

3. Memory is not utilized efficiently.

4. Linear data structures are easy to implement.

5. Example: Array, Stack, Queue, Linked list etc.

Non-Linear Data Structures

1. In non-linear data structures, elements are arranged in hierarchical manner.

2. Elements are based on multiple levels.

3. Memory is utilized efficiently.

4. Non-linear data structures are complex.

5. Example: Tree, Graph, Hash Table etc.

Q # 03. Name some commonly used abstract data types.

Abstract Data Types (ADT) :
Abstract Data Types (ADT) defines a set of data and operations but no implementation. It only mentions what operations are to be performed but not how they are going to be implemented. Some commonly used abstract data types are list as ADT, stack as ADT and queue as ADT.

Q# 04. What do you understand by the term Algorithm analysis? Discuss.

Algorithm Analysis:
Algorithm is a step by step sequence of instructions to solve a specific problem. But different algorithms solve problems differently. Some work faster while others take more computation time. In order to find the efficiency and accuracy of an algorithm we analyze the algorithm on the basis of its time and space complexity. The process of analyzing an algorithm is known as Asymptotic Analysis. It refers to computing the running time of any operation in mathematical units of computation. Time required to solve a specific problem usually falls under three types i.e. Best Case, Average Case and Worst Case.
Best Case refers to the minimum time required for execution of a program. It is also known as lower bound of an algorithm and is represented by Ω notation.
Worst Case refers to the maximum time required for execution of a program. It is also known as upper bound of an algorithm and is represented by O notation.
Average Case refers to the average time required for a program execution. It is represented by θ notation.

Lab # 2

Q # 01. Let an array named LA consist of 4 elements 2,4,6 & 8 . write down the code to traverse(or print) all elements in the array.

CODE:
```python
import numpy as np
LA= np.array([2,4,6,8])
n= len(LA)
def traversal(arr):
   start=0
   while start<n:
      print(arr[start])
      start=start+1
traversal(LA)
```

OUTPUT:
2
4
6
8

Q # 02. Use insertion algorithm to add an element (Give implementation)
i. 1 at index 0
CODE:
```python
import numpy as np
LA= np.array([2,4,6,8,0])
k=0
item=1
def insertion(arr):
   n = 4
   j=n-1
   n = n + 1
   while j>=k:
      arr[j+1]=arr[j]
      j=j-1
   arr[k]=item
   n=n+1
insertion(LA)
print(LA)
```

OUTPUT:
[1 2 4 6 8]

ii. 5 at index 2
CODE:
```
import numpy as np
LA= np.array([2,4,6,8,0])
k=2
item=5
def insertion(arr):
    n = 4
    j=n-1
    n = n + 1
    while j>=k:
        arr[j+1]=arr[j]
        j=j-1
    arr[k]=item
    n=n+1
insertion(LA)
print(LA)
```

OUTPUT:
[2 4 5 6 8]

iii. 3 at index 4
CODE:
```
import numpy as np
LA= np.array([2,4,6,8,0])
k=4
item=3
def insertion(arr):
    n = 4
    j=n-1
    n = n + 1
    while j>=k:
        arr[j+1]=arr[j]
        j=j-1
    arr[k]=item
    n=n+1
```

insertion(LA)
print(LA)

OUTPUT:
[2 4 6 8 3]

Q # 03. If the size of the given array is 4 then calculate the Time complexity of insertion
algorithm when ,
i. Insert element at the beginning of array
ii. Insert element at the end of array
iii. Insert element in the middle of array

Answer:

Time Complexity at the beginning of array	O(N)
Time Complexity at the end of array	O(N)
Time Complexity at middle of array	O(N)

Q # 04. If the size of given array is 4 then calculate the Time complexity of deletion
algorithm when ,
i. Insert element at the beginning of array
ii. Insert element at the end of array
iii. Insert element in the middle of array

Answer:

Time Complexity at the beginning of array	O(N)
Time Complexity at the end of array	O(N)
Time Complexity at middle of array	O(N)

Lab # 3

Q1: Assume array[]= {2,4,6,8,}. Give implementation of Linear search algorithm to ;

i. find element 8 in array

ii. find element 3 in array

CODE:

Implementation of the linear search algorithm

```python
def search(arr, x):

        for i in range(len(arr)):

        if arr[i] == x:

         return i

        else:

        return -1


print("The targeted element is at index: ", search([2, 4, 6, 8], 8))

print("The targeted element is at index: ", search([2, 4, 6, 8], 3))
```

OUTPUT:

```
"D:\Python Practice\venv\Scripts\python.exe" "D:/Python Practice/dsa labs.py"
The targeted element is at index:  3
The targeted element is at index:  -1

Process finished with exit code 0
```

Q2: let assume A[]= {2,5,5,5,6,6,8,9,9,9} sorted array of integers containing duplicates, apply binary search algorithm to Count occurrences of a number provided, if the number is not found in the array report that as well

Solution:

CODE: # Program to count occurrences of an element

A recursive binary search function. It returns location of x in given array arr[l..r]

is present, otherwise -1

```
def binarySearch(arr, l, r, x):

        if (r < l):

                return -1


        mid = int( l + (r - l) / 2)
```

If the element is present at the middle itself

```
        if arr[mid] == x:

                return mid
```

If element is smaller than mid, then it can only be present in left subarray

```
        if arr[mid] > x:

                return binarySearch(arr, l,mid - 1, x)
```

Else the element can only be present in right subarray

```
        return binarySearch(arr, mid + 1, r, x)
```

Returns number of times x occurs in arr[0..n-1]

```
def countOccurrences(arr, n, x):

        ind = binarySearch(arr, 0, n - 1, x)
```

```python
        # If element is not present
        if ind == -1:
            return 0
    # Count elements on left side.
        count = 1
        left = ind - 1
        while (left >= 0 and arr[left] == x):
            count += 1
            left -= 1
    # Count elements on right side.
        right = ind + 1;
        while (right < n and arr[right] == x):
            count += 1
            right += 1


        return count


# Driver code
arr = [ 2,5,5,5,6,6,8,9,9,9 ]

n = len(arr)

x = int(input("Input: "))
```

print("Element", x, "occurs", countOccurrences(arr, n, x) ,"times")

OUTPUT:

```
"D:\Python Practice\venv\Scripts\python.exe" "D:/Python Practice/dsa labs.py"
Input: 5
Element 5 occurs 3 times

Process finished with exit code 0
```

Q3: Compare linear & binary Search algorithms on the basis of time complexity, which one is better in your opinion? Discuss.

Answer: The time complexity of the linear search is O(N) while binary search has $O(log_2N)$. The best-case time in linear search is for the first element i.e., O(1). As against, in binary search, it is for the middle element, i.e., O(1). In the linear search, worst case for searching an element is N number of comparison. In contrast, it is log2N number of comparison for binary search. Linear search can be implemented in an array as well as in linked list whereas binary search can not be implemented directly on a linked list. Both linear and binary search algorithms can be useful depending on the application. When an array is the data structure and elements are arranged in sorted order, then binary search is preferred for quick searching. If the linked list is the data structure regardless of how the elements are arranged, linear search is adopted due to unavailability of direct implementation of the binary search algorithm. Hence, there is a requirement to design the variation of the binary search algorithm that can work on a linked list too because the binary search is faster in execution than a linear search.

Lab # 4

Q1: Give implementation of bubble sort algorithm let's assume Array "A" is consisting of 6

elements which are stored in descending order

Answer: # Python program for implementation of Bubble Sort

```
def bubbleSort(arr):

    n = len(arr)


    # Traverse through all array elements
    for i in range(n - 1):
    # range(n) also work but outer loop will repeat one time more than needed.
    # Last i elements are already in place
    for j in range(0, n - i - 1):


    if arr[j] > arr[j + 1]:
        arr[j], arr[j + 1] = arr[j + 1], arr[j]



arr = [70, 63, 50, 41, 30, 10]


bubbleSort(arr)
```

print("Sorted array is:")

for i in range(len(arr)):

      print("%d" % arr[i])

OUTPUT:

```
"D:\Python Practice\venv\Scripts\python.exe" "D:/Python Practice/dsa labs.py"
Sorted array is:
10
30
41
50
63
70

Process finished with exit code 0
```

Q2: Re-write bubble sort algorithm for a recursive function call, also analyze the time complexity of recursive bubble sort algorithm.

Answer: //Recursive Bubble Sort

let recursiveBubbleSort = (arr, n = arr.length) => {

 //If there is only single element

 //the return the array

 if(n == 1){

      return arr;

 }

 //Swap the elements by comparing them

```
for(let j = 0; j < n - 1; j++){

        if(arr[j] > arr[j + 1]){

        [arr[j], arr[j+1]] = [arr[j+1], arr[j]];

        }

    }


    //Recursively call the function to sort.

    return recursiveBubbleSort(arr, n-1);

}
```

Time complexity is $O(n \char`^ n) = O(n \char`^ 2)$.

Q3: Implement the Quick sort Algorithm on Array A[]={ 9,7,5,11,12,2,14,3,10,6}

Solution:

CODE:

```
import random

def QuickSort(A, low, high):

        if low < high:

        pivot = Partition(A, low, high)

        QuickSort(A, low, pivot - 1)

        QuickSort(A, pivot + 1, high)


def Partition(A, low, high) :

        pivot = low + random.randrange(high - low + 1)
```

```python
        swap(A, pivot, high)

        for i in range(low, high):

        if A[i] <= A[high]:

        swap(A, i, low)

        low += 1


        swap(A, low, high)

        return low


def swap(A, x, y):

        temp = A[x]

        A[x] = A[y]

        A[y] = temp


A = [9,7,5,11,12,2,14,3,10,6]

QuickSort(A, 0, len(A) - 1)

print(A)
```

OUTPUT:

```
"D:\Python Practice\venv\Scripts\python.exe" "D:/Python Practice/dsa labs.py"
[2, 3, 5, 6, 7, 9, 10, 11, 12, 14]

Process finished with exit code 0
```

Q4: Compare the two sorting algorithms (discuss in this session) and discuss their advantages and disadvantages.

ANSWER:

Comparison between Bubble and Selection Sort Algorithms

Bubble sort is a sorting algorithm that compared adjacent element and then swaps. While, Selection sort is a sorting algorithm that selects the largest number and swap with the last number. Bubble sort is not good in terms of efficiency as compared to selection sort. Bubble sort use exchanging method while other uses selection method. The complexity of bubble sort is O(n), while Selection Sort complexity is O(n^2).

Lab # 5

Q1: Give implementations of Push & Pop Algorithms with underflow & Overflow exceptions?

CODE:

```python
class Stack(object):

    def __init__(self, limit=10):
        self.stk = []
        self.limit = limit


    def isEmpty(self):
        return len(self.stk) <= 0


    def push(self, item):
        if len(self.stk) >= self.limit:
            print("Stack Overflow!")
        else:
            self.stk.append(item)
    print("Stack after Push', self.stk")


    def pop(self):
        if len(self.stk) <= 0:
            print("Stack Underflow!")
            return 0
```

```python
        else:
         return self.stk.pop()


    def peek(self):
    if len(self.stk) <= 0:
     print("Stack Underflow!")
    return 0
    else:
     return self.stk[-1]


    def size(self):
    return len(self.stk)



stk = Stack(5)

stk.push(-5)

stk.push(1)

stk.push(21)

stk.push(14)

stk.push(31)

stk.push(19)

stk.push(3)
```

stk.push(99)

print(stk.peek())

print(stk.pop())

print(stk.peek())

print(stk.pop())

OUTPUT:

```
"D:\Python Practice\venv\Scripts\python.exe" "D:/Python Practice/dsa labs.py"
Stack after Push', self.stk
Stack Overflow!
Stack Overflow!
Stack Overflow!
31
31
14
14

Process finished with exit code 0
```

Q2: What should be the time complexity of algorithms (discuss in this session) in your opinion?

Discuss.

ANSWER:

Let n be the number of elements in the stack. The complexities of stack operations with this representation can be given as:

Time Complexity of Push()	O(1)
Time Complexity of Pop()	O(1)
Time Complexity of Size()	O(1)

Time Complexity of IsEmpty()	O(1)

Q3: Write an algorithm to Reverse a string using a stack data structure also gives implementation.

SOLUTION:

Input:

```python
# Python program to reverse a string using stack

def createStack():

        stack = []

        return stack


def size(stack):

        return len(stack)


def isEmpty(stack):

        if size(stack) == 0:

        return true

# It increases size by 1

def push(stack, item):

        stack.append(item)


# It decreases size by 1
```

```python
def pop(stack):

    if isEmpty(stack): return

    return stack.pop()



def reverse(string):

    n = len(string)


    stack = createStack()


    # Push all characters of string to stack

    for i in range(0, n, 1):

    push(stack, string[i])


    string = ""


    # Pop all characters of string and

    # put them back to string

    for i in range(0, n, 1):

    string += pop(stack)


    return string
```

string = "Hello World!"

string = reverse(string)

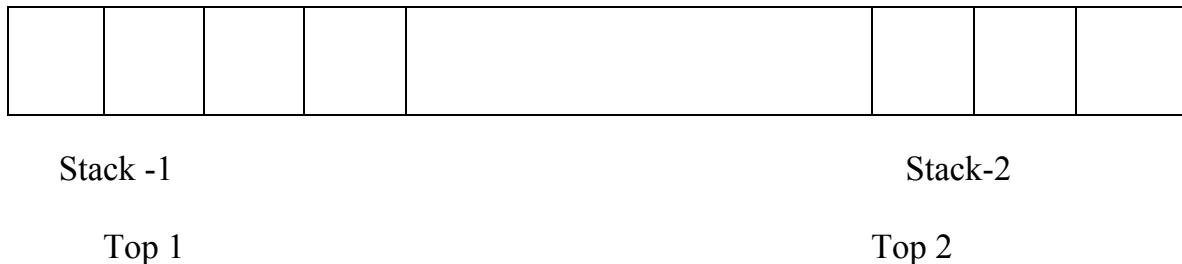print("Reversed string is: " + string)

Output:

```
"D:\Python Practice\venv\Scripts\python.exe" "D:/Python Practice/dsa labs.py"
Reversed string is: !dlroW olleH

Process finished with exit code 0
```

Q4: Write an algorithm to implement two stacks in a single array.

ANSWER:

Stack -1   Stack-2

Top 1   Top 2

ALGORITHM:-   Start two indexes one at the left end and other at the right end.

- The left index simulates the first stack and right index simulates the second stack.

- If we want to push an element into first stack then put elements at left stack.

- Similarly, if we want to push an element into second stack the put the element at the right index.

- The first stack grows towards the right, and the second stack grows towards the left.

Lab # 6

Q1: Solve the following postfix expressions via algorithm.

      i.   P: 5 ,6,2,+,*, 12,4,/,-

      ii.   . 2,3,^,5,2,2,^,*,12,6,/,-,+

Solution:

i)   5,6,2,+,*,12,4,/,-

Symbols Scanned	Stack
5	5
6	5,6
2	5,6,2
+	5,8
*	40
12	40,12
4	40,12,4

/	40,3
-	37

ii) 2,3,^,5,2,2,^,*,12,6,/,-,+

Symbols Scanned	Stack
2	2
3	2,3
^	8
5	8,5
2	8,5,2
2	8,5,2,2
^	8,5,4
*	8,20
12	8,20,12
6	8,20,12,6
/	8,20,2

-	8,18
+	26

Q2: Implement the algorithm to convert infix **(A +(B*C –(D / E ^ F) *G) *H)** expression to equivalent postfix form.

Solution:

SYMBOL SCANNED	STACK	EXPRESSION P
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
-	(+(*-	ABC*
(	(+(*-(	ABC*D
D	(+(*-(	ABC*D
/	(+(*-(/	ABC*D

E	(+(*-(/	ABC*DE
^	(+(*-(/^	ABC*DE
F	(+(*-(/^	ABC*DEF
)	(+(-	ABC*DEF^/
*	(+(-*	ABC*DEF^/
G	(+(-*	ABC*DEF^/G
)	(+	ABC*DEF^/G*-
*	(+*	ABC*DEF^/G*-
H	(+*	ABC*DEF^/G*-H
)		ABC*DEF^/G*-H*+

Q3: Write algorithm to solve prefix expression also gives implementation.

ALGORITHM:

Step 1: Put the cursor at the end and start scanning from the right.

Step 2: If the character occurred at P is operand then push it to stack

Step 3: If the character occurred at P is operator then pop the two elements from stack and perform the operation occurring to the operator occurred and finally push the result back to stack

Step 4: Decrement P by 1 and go to step 2 as long as there is character available to be scanned in stack

Step 5: The result is stored at the top of stack so return it

Step 6: End

CODE:

```python
def is_intNo(c):
    return c.isdigit()

def evaluate_No(expression):
    stack = []
    for i in expression[::-1]:
        if is_intNo(i):
            stack.append(int(i))

        else:
            x1 = stack.pop()
            x2 = stack.pop()
            if i == '+':
                stack.append(x1+x2)
            elif i == '-':
                stack.append(x1-x2)
            elif i == '*':
                stack.append(x1*x2)
            elif i == '/':
                stack.append(x1/x2)
    return  stack.pop()
test_expression = "+9*26"
print("The solution of infix expression ",test_expression,"is ",evaluate_No
    (test_expression))
```

OUTPUT:

```
('The solution of infix expression ', '+9*26', 'is ', 21)
```

Lab # 7

1. Write Algorithms for Enqueue and dequeue operations in a circular queue

This algorithm is used to insert or add items into a circular queue.

Enqueue(queue, front, rear, max, count, item)

      1.   if (count = max) then

      a. display "queue overflow";

      b. return;

      2.   otherwise

      a. if (rear = max) then

      i.   rear: = 1;

      b. otherwise

      i.   rear:= rear + 1;

      c. queue(rear):= item;

      d. count:= count + 1;

      3.   Return;

This algorithm is used to remove or delete items from a circular queue.

dequeue(queue, front, rear, count, item)

   1.   if (count = 0) then

      a. display "queue underflow";

      b. return;

   2.   otherwise

      a. item := queue(front)l

      b. if (front =max) then

      i.   front := 1;

      c. otherwise

      i.   front := front + 1;

    d. count := count + 1;

  3.   return;

2. Give implementation of queue data structure using two stacks (let S1 & S2 be the two stacks for push and pop operations respectively).

CODE

```python
class Queue(object):

    def __init__(self):

        self.S1 = [] self.S2 = []

    def enqueue(self, element):

        self.S1.append(element)

    def dequeue(self):

        if not self.S2:

            while self.S1:

                self.S2.append(self.S1.pop())

        return self.S2.pop()

q = Queue()for i in range(5):

q.enqueue(i)

for i in range(5):

    print (q.dequeue())
```
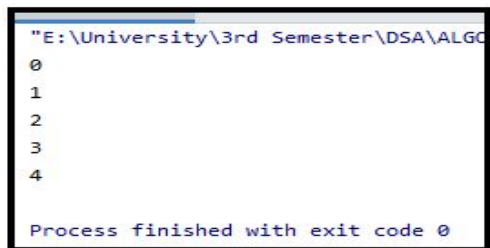
OUTPUT


```
"E:\University\3rd Semester\DSA\ALGO
0
1
2
3
4

Process finished with exit code 0
```

Lab # 8

Q # 01. Calculate the Time complexity of Fibonacci and factorial recursive algorithms ,Also give the upper bound of both algorithms.

Time Complexity : (factorial recursive algorithm)

T(n)=T(n-1)+3   à I   //if n>0

T(0)= 1   à //if n=0

T(n)=T(n-1)+3

T(n)=(T(n-2)+3) +3à T(n-2)+6

T(n)=((T(n-3)+3)+3)+3àT(n-3)+9

Generally

T(n)= T (n-k)+3K

For k=n => n-k=0

=T(0)+3n

T(n)=1+3n

Upper Bound :

O(n)= n

Time Complexity : (Fibonacci recursive algorithm)

T(n)=T(n-1)+T(n-2)+4 ; if n>1

T(0)= 1 ; if n<=1

Since T(n-1)takes more time so we assume here T(n-2) almost takes same time, so we rewrite above equation 1 as:

T(n)=T(n-1)+T(n-1)+4

T(n) =2T(n-1)+4 or

T(n)= 2T(n-1)+ C àii

T(n)=2 (2(T(n-2)+C)+C à 4T(n-2)+3C

T(n)= 4T(n-2)+3C

T(n)=4 (2T(n-3)+C) +3C

T(n)=8T(n-3)+7C

similarly:

T(n)=8(2T(n-4)+C) +7C

T(n)=16T(n-4)+15C

T(n)=16T(n-4)+15C

Generally,

T(n)=2^k   T(n-k)+(2^k -1)C

 at k=n we have n-k=0

T(n) =2^n T(0)+(2^n -1)C

T(n)= 2^n (1)+(2^n -1)C

T(n)=2^n +2^n C -C=> 2^n (1+C)-C

Upper Bound :

T(n) = 2^n

Q # 02. Write iterative factorial and Fibonacci algorithms ,also analyze the time complexity.

Factorial Algorithm (iterative)

Start:

1.   SET fact(variable) = 1

2.   For i in range n

3.   SET fact += fact * n

3. Print fact

End

Time Complexity:

Statement	Operation	Iteration	Subtotal
1	1	1	1
2a	1	1	1
2b	1	n+1	n+1
2c	1	n	n
3	3	1	1
			2n+3

Upper Bound :

$T(n) = n$

Fibonacci Algorithm (iterative)

Start:

1. Take input value n

2. Set n0 = 0 and n1=1

3. If n<0 return 0

4. Else if n==1 return 1

5. Else

    For i in range n

    SET total = n0 + n1

    Exchange values of n0 , n1 and assign total to n1

6. Print total

End

Time Complexity:

Statement	Operation	Iteration	Subtotal
1	2	1	2
2a	2	1	2
2b	1	1	1
3a	1	1	1
3b	3	1	1
4a	1	1	1
4b	1	n+1	n+1
4c	1	n	n
5	2	n	2n
6	1	n	n
7	1	n	n
8	1	1	1
			6n+10

Upper Bound :

T(n) = n

Q # 03. Write a recursive algorithm for Tower of Hanoi Puzzle also calculate its upper bound (also give implementation) .

Algorithm

START

Def TOH(n, srce, aux, des)

  IF n>= 1, THEN

  Hanoi(n- 1, srce, dest, aux)   // Step 1

   move disk from source to dest   // Step 2

Hanoi(n- 1, aux ,srce , dest)   // Step 3

END

Time Complexity:

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

$T(n-1) = 2\ T(n-2)+1$ –eq-2

$T(n-2) = 2\ T(n-3)+1$ --eq-3

Put value of $T(n-2)$ in eq–2 with help of eq-3

$T(n-1) = 2(2\ T(n-3)+1)+1$ --eq--4

Put value of $T(n-1)$ in equation-1 with help of eq-4

$T(n) = 2(2(2\ T(n-3)+1)+1)+1$

$T(n) = 2^3(T(n-3)+2^2+2^1+1$

After Generalization :

$T(n) = 2^k T(n-k) + 2^{(k-1)} + 2^{(k-2)} + 2 + \ldots\ldots\ldots + 2^2 + 2^1 + 1$

Base condition $T(0) == 1$

$n - k = 0$

$n = k;$

put, $k = n$

$T(n) = 2^n T(0) + 2^{(n-1)} + 2^{(n-2)} + 2 + \ldots\ldots\ldots + 2^2 + 2^1 + 1$

It is GP series, and sum is $2^{(n+1)} - 1$

Upper Bound :

$T(n) = 2^n$

Implementation:

CODE:

```python
def TOH(n,a,b,c):
        if (n>=1):
        TOH(n-1,a,c,b)
        print(f"Moving disk {n} from {a} to {c}  ")
        TOH(n-1,b,a,c)
TOH(3,"A","B","C")
```

Output:

Moving disk 1 from A to C

Moving disk 2 from A to B

Moving disk 1 from C to B

Moving disk 3 from A to C

Moving disk 1 from B to A

Moving disk 2 from B to C

Moving disk 1 from A to C

Lab # 9

1. Construct a binary tree (initially empty) , insert nodes 15,10,20,25,8,12 with the help of following definition of Node in BST
Struct BSTNode{
 Int data;
 BSTNode* left;
 BSTNode* right; } BSTNode* rootPtr;

CODE:

```
class Node:

    def _init_(self, key):
        self.key = key
        self.left = None
        self.right = None


def insert(node, key):


    if node is None:
        return Node(key)

    if key < node.key:

        node.left = insert(node.left, key)
    else:
        node.right = insert(node.right, key)


        return node


root = None
root = insert(root,15)
root = insert(root, 10)
root = insert(root, 20)
root = insert(root, 25)
root = insert(root, 8)
root = insert(root, 12)
```

2. Delete One left-child-node and one right-child- node from above tree. Analyze the chang
occur in the tree after deletion.

CODE:

```python
def minValueNode(node):
    current = node

    while(current.left is not None):
        current = current.left

    return current

def deleteNode(root, key):

    # Base Case

    if root is None:
        return root

    if key < root.key:

        root.left = deleteNode(root.left, key

    elif(key > root.key):

        root.right = deleteNode(root.right, key)

    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 = minValueNode(root.right)

        root.key = temp.key

        root.right = deleteNode(root.right, temp.key)

    return root
```

```
root = None
root = delete(root, 15)
root = delete(root, 10)
root = delete(root, 20)
root = delete(root, 25)
root = delete(root, 8)
root = delete(root, 12)

root = delete(root, 12)
```

Q3: From the figure below, identify i. root ii. Height of tree iii. Degree iv. Size v. leafnode(s)

      i.   Root

      ii.   Height of tree

      iii.   Degree

      iv.   Size

      v.   Leaf node(s)

Solution:

ð Root = The distinguished element that is the origin of the tree is A.

ð Height of Tree = Number of edges in longest path from root to leaf node is Three.

ð Degree of tree = The maximum numbers of children of a node is called the degree of a binary tree is Three.

ð Size of tree = Number of all nodes is Nine.

ð Leafnodes of Tree = Number of all nodes are D,E.F,G,I.

Q4: Draw a tree from the following representation: and identify i. Child of H ii. Height of Tree iii. Leaf node(s) iv. Parent of A.
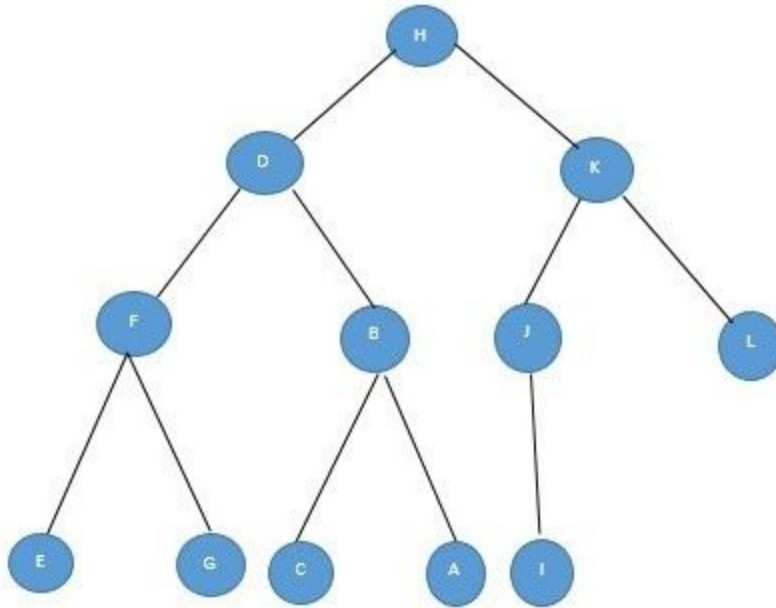
 D->F

D->B

K->J

K->L

B->A

B->C

H->D

H->K

F->E

F->G

J->I

Solution:

ð Child of H are D, K.

ð Height of Tree is 3.

ð Leaf node(s) are E,G,C,A,I,L

ð Parent of A is B.

Q5: write down the applications of Tree data structure.

Solution:

ð A tree is a nonlinear data structure used to represent entities that are in some hierarchical relationship. It is defined as a collection of nodes. Nodes represent values and nodes are connected by edges.

APPLICATIONS:

ð Tree data structure is used to store some form of information that are naturally in the form of a hierarchy.

ð Tree data can also be used to organize keys. It is used to search key faster than liked list but it is slower than arrays.

ð Tree works very efficiently to manipulate hierarchical data and sorted list.

ð Tree data structure also makes information very easy to search.
ð) Tree data structure is used as a work flow for composing digital images. Two of its types B- AND B+ tree can be used in indexing of a database.

ð Heap is one of its kinds which are implemented using array and used to implement prior queues
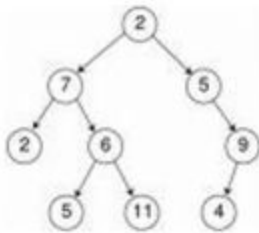
Lab # 10

Q1: Traverse the following binary trees using the pre, in, and post order traversal methods
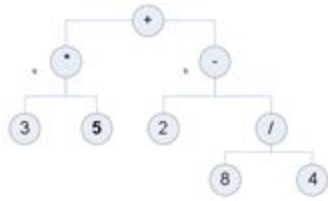
Post order: 3 5 * 2 8 4 / - +

Pre order: + * 3 5 - 2 / 8 4

In order: 3 * 5 + 2 - 8 / 4

Post order: 2 5 11 6 7 4 9 5 2

Pre order: 2 7 2 6 5 11 5 9 4

In order: 2 7 5 6 11 2 5 4 9

Q2 Implement all three tree traversal algorithms, also determine the worst time complexity

CODE

class Node:

    def __init__(self,key):

    self.left = **None**

    self.right = **None**

    self.val = key

```python
def Preorder(root):

    if root:

        print(root.val,end =" "),

        Preorder(root.left)

        Preorder(root.right)

def Postorder(root):

    if root:

        Postorder(root.left)

        Postorder(root.right)

        print(root.val,end =" "),

def Inorder(root):

    if root:

        Inorder(root.left)

        print(root.val,end =" "),

        Inorder(root.right)

root = Node(2)

root.left = Node(7)

root.right = Node(5)

root.left.left = Node(2)

root.left.right=Node(6)

root.left.right.left=Node(5)

root.left.right.right=Node(11)

root.right.right = Node(9)

root.right.right.left = Node(4)

print("\n""Preorder traversal of binary tree is",end =" ")
```

printPreorder(root)

print("\n")

print("Inorder traversal of binary tree is",end =" ")

printInorder(root)

print("\n")

print("Postorder traversal of binary tree is",end =" ")

printPostorder(root)

OUT PUT

```
"E:\University\3rd Semester\DSA\ALGO\venv\Scripts\python.exe

Preorder traversal of binary tree is 2 7 2 6 5 11 5 9 4

Inorder traversal of binary tree is 2 7 5 6 11 2 5 4 9

Postorder traversal of binary tree is 2 5 11 6 7 4 9 5 2
```

worst time complexity

ALGORITHM:

PreOrder(root)

If root = null   ------->   T (1)

        Return   ----->   T (1)

else: ----->   T (1)

Write: root;   ----->   T (1)

PreOrder(root->left)   -----> T(n/2)

PreOrder(root->right)   ----->   T(n/2)

There are four basic operations which takes constant unit of time. So, their time complexity is O(n) Preorder function calls for left subtree and right subtree separately and each time divided the tree by half.

GENERAL EQUATION:

$T(n)=T(n/2) +T(n/2) +O (1)$

$T(n)= 2T(n/2) + O (1)$

$T (1) =1$

$T(n)= 2T(n/2) + O (1)$ eq (1)

$T(n)= 2T(n/2) + O (1)$

$T(n/2) = 2[2T(n/4)] + O (1)$

$T(n/2) = 4T(n/4) + O (1)$ eq (2)

$T(n/3) = 2[4T(n/8)] + O (1)$

$T(n/3) = 8T(n/8) + O (1)$ eq (3)

Generally,

$T(n)=2^kT(n/2^k) + O (1)$ eq(A)

At some point, number of nodes reduced to 1;

$T(n)=T (1)$

$n/2^k=1$

$n=2^k$

Taking log on both sides

$\log(n)=\log(2^k)$

$\log(n)=k\log2$ [log2=1]

$\log(n)=k$

Equation becomes A:

$T(n)=2^{\log(n)}T(n/2^{\log(n)}) + O (1)$ eq(B)

Let $t= 2^{\log(n)}$ eq(a)

Taking log on both sides of eq(a)

$\log(t)=\log(2^{\log(n)})$

log(t)=log(n)*log (2)　　　　　[log2=1]

log(t)=log(n)

Taking anti log on both sides

Log^-1(log(t)) =log^-1(log(n))

t=n

From eq(a)

2^log(n)=n

Equation (B) becomes:

T(n)=nT(n/n) + O (1)

T(n)=nT(n/n) + O (1)

T(n)=nT (1) +O (1)　　　　　[T (1)=1]

T(n) = O(n)

Q3 Gives the implementation of Print all nodes of a perfect binary tree in Top-to-Down order.

Code:

```python
from collections import deque
class Node:

        def __init__(self, key=None, left=None, right=None):

            self.key = key

            self.left = left

            self.right = right

def Nodes(root):

        if root is None:

        return

        print(root.key, end=' ')

        q1 = deque()
```

```python
        q2 = deque()
        q1.append(root.left)
        q2.append(root.right)
        while q1:
            n = len(q1)
            for _ in range(n):
                x = q1.popleft()
                print(x.key, end=' ')
                if x.left:
                    q1.append(x.left)
                if x.right:
                    q1.append(x.right)
                y = q2.popleft()
                print(y.key, end=' ')
                if y.right:
                    q2.append(y.right)
                if y.left:
                    q2.append(y.left)
if __name__ == '__main__':
    root = Node(1)
    root.left = Node(2)
    root.right = Node(3)
    root.left.left = Node(4)
    root.left.right = Node(5)
    root.right.left = Node(6)
```
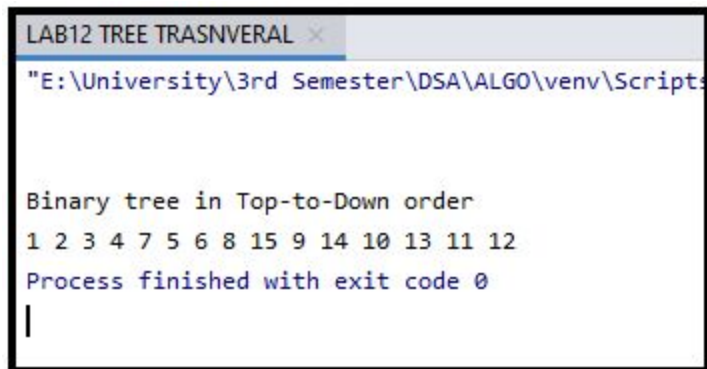
```python
        root.right.right = Node(7)

        root.left.left.left = Node(8)

        root.left.left.right = Node(9)

        root.left.right.left = Node(10)

        root.left.right.right = Node(11)

        root.right.left.left = Node(12)

        root.right.left.right = Node(13)

        root.right.right.left = Node(14)

        root.right.right.right = Node(15)

print("\n")

print("Binary tree in Top-to-Down order")

Nodes(root)
```

OUTPUT


```
LAB12 TREE TRASNVERAL ×
"E:\University\3rd Semester\DSA\ALGO\venv\Scripts


Binary tree in Top-to-Down order
1 2 3 4 7 5 6 8 15 9 14 10 13 11 12
Process finished with exit code 0
```

Lab # 11

1. Gives the algorithms for adjacency list and adjacency matrix methods .also determine their time complexity.

Algorithm:(adjacency matrix)

Variables:

mè no. of edges

u,vè just a variable

nè no. of rows and columns of a matrix

A è a matrix

Start

1. Take input from user of value of m, n

2. For i in range n

   For j in range n

    Set A[i][j] = 0

3. For i in range m

   Take input of value of u and v

   Set A[u][v]=1

   Set A[v][u]=1

4. For i in range n

   For j in range n

    Print A[i][j]

End

Algorithm:(adjacency list)

Variables:

mè no. of edges

u, vè just an adjacency list

nè vertices

A è a matrix

Start

1. Take input from user of value of m, n

2. For i in range n:

>   Set A[i] = []

3. For i in range m

>   Take input of value of u and v

>   Set A[u] = v

>   Set A[v] = u

4. For i in range A

>   Print A[i]

End

Time Complexity :(matrix)

Assuming the graph has vertices, the time complexity to build such a matrix is n^2

To fill every value of the matrix we need to check if there is an edge between every pair (v1,v2) of vertices. The amount of such pairs of n given vertices is n(n+1)/2. That is why the time complexity of building the matrix is $O(n^2)$.

Statement	Operation	Iteration	Total
1	2	1	2
2	7	n^2	7n^2
3	7	n	n
4	7	n^2	7n^2
			14n^2 + n +2

(list)

If m is the number of edges in a graph, then the time complexity of building such a list is O(m).

Statement	Operation	Iteration	Total
1	2	1	2
2	4	n	4n
3	7	n	n
4	4	n	4n
			9n+2

Implementation (adjacency matrix)

n=4

__g =[[0 for x in range(4)] for y in range(4)]

def adEdge(x,y):

    if (x == y):

    print("Same Vertex !")

    if (x >= n) or (y >= n):

    print("Vertex does not exists !")

    else:

```
        __g[y][x] = 1

        __g[x][y] = 1


adEdge(0,1)

adEdge(0,2)

adEdge(1,2)

adEdge(2,3)

for i in range(4):

        print()

        for j in range(4):

        print("", __g[i][j], end ="")
```

Output:

```
 0  1  1  0

 1  0  1  0

 1  1  0  1

 0  0  1  0
```

Implementation (adjacency list)

```
class adjNode:

        def __init__(self,node):

        self.nodes=node

        self.adjlist={}

        for node in self.nodes:
```

```
        self.adjlist[node]=[]

        def ad_edge(self,u,v):

        self.adjlist[u].append(v)

        self.adjlist[v].append(u)

        def prin(self):

        for node in self.adjlist:

        print(f"Vertex {node} --> {self.adjlist[node]}")

node = [0,1,2,3]

g = adjNode(node)

g.ad_edge(0,1)

g.ad_edge(0,2)

g.ad_edge(2,1)

g.ad_edge(2,3)

g.prin()
```

Output:

Vertex 0 → [1 , 2]

Vertex 1 → [0 , 2]

Vertex 2 → [0 , 1 , 3]

Vertex 3 → [2]

Q # 03. Differentiate between graph and Tree data structure.

Graph	Tree

A graphical representation of nonlinear data where data is denoted by nodes connected through edged	Tree is also used to represents the nonlinear data but in context of hierarchy
No unique node of root is there	A unique node is there namely root
Usually a cycle can be formed in a graph	There is no certain cycles are there
As Graph did not represent the data is hierarchical manner so there is no parent child relation between data representation	Tree data is represented in hierarchical manner so parent to child relation exists between the nodes
Main use of graphs is colouring and job scheduling	On other hand Main use of trees is for sorting and traversing.
Applications: For finding shortest path in networking graph is used	Applications: For game trees, decision trees, the tree is used.

Lab # 12

1. give Implementation of depth & breadth first search algorithms on graph (lab 11 Q2)

Breadth First Search Implementation:

<u>**Code :**</u>

```python
from collections import deque

def Breadth(graph, fvertex):

        visitedNodes=set()

        q=deque([fvertex])

        visitedNodes.add(fvertex)

        while q:

        vertex = q.popleft()

        print(vertex , " ", end="")

        for AdjNode in graph[vertex]:

        if AdjNode not in visitedNodes:

                visitedNodes.add(AdjNode)

            q.append(AdjNode)

if __name__ == '__main__':

        graph = {0: [1, 2],

        1: [2],

        2: [3],

        3: [2]}

        print("The Breadth First Search Traversal is  : ")

        Breadth(graph, 0)
```

<u>**Output:**</u>

```
"E:\University\3rd Semester\DSA\ALGO\venv\Sc
The Breadth First Search Traversal is  :
0  1  2  3
Process finished with exit code 0
```

Depth First Search Implementation:

<u>CODE:</u>

def Depth(visitednodes,graph, vertex):

  if vertex **not in** visitednodes:

      print(vertex,**" "**,end=**" "**)

      visitednodes.add(vertex)

      for AdjNode **in** graph[vertex]:

      Depth(visitednodes,graph,AdjNode)

if __name__ == **'__main__'**:

      graph = {0: [1, 2],

      1: [2],

      2: [3],

      3: [2]}

      visitednodes=set()

      print(**"The Depth First Search Traversal is : "**)

      Depth(visitednodes,graph, 0)

<u>OUTPUT</u>

```
"E:\University\3rd Semester\DSA\ALGO\venv\Scripts\pyt
The Depth First Search Traversal is  :
0   1   2   3
Process finished with exit code 0
```

2. Use BFS (diagram below) to Find the path between given vertices in a directed graph

CODE

```python
from collections import deque
class Graph:
        def __init__(self, edges, nodes):
        self.ad_list = [[] for _ in range(nodes)]
        for (source, destination) in edges:
        self.ad_list[source].append(destination)
def path(graph, source, destination, found):
        found[source] = True
        q1 = deque()
        q1.append(source)
        while q1:
            vertex = q1.popleft()
        if vertex == destination:
            return True
        for val in graph.ad_list[vertex]:
        if not found[val]:
                found[val] = True
                q1.append(val)
        return False
if __name__ == '__main__':
        edges = [
        (0, 3), (1, 0), (1, 2), (1, 4), (2, 7), (3, 4),
        (3, 5), (4, 3), (4, 6), (5, 6), (6, 7)
        ]
        nodes = 8
        g1 = Graph(edges, nodes)
        found = [False] * nodes
        source = 0
        destination = 7
        if path(g1, source, destination, found):
        print("Path is found between ", source, "and", destination, " vertices")
        else:
        print("Path is not found between ", source, "and", destination, " vertices")
```

OUTPUT

3.Write algorithm to find minimum no. of throws required to win snake & Ladder game by using BFS approach , also analyze its time complexity .(Give implementation as well)

Algorithm

This algorithm is used to find the minimum number of throws to win the Snake and Ladder game.

Step 0 Initialize a graph with N nodes (where N=no. of small boxes on board). It has

A.   No. of nodes

B.   List of edges

1 procedure Breadth(graph,nodes,source)

2   set q=deque()

3   set found=false*nodes+1

4   set found[source]=true

5   append value of source in q

6   repeat steps 6 to 15 until q is empty

7   vertex,min_dist=q.popleft()

8   if vertex is equal to nodes

9   print (min_dist)

10   break

11   Repeat steps 11 to 15 till the values in graph[vertex]

12   If value in graph[vertex] is false

13   Found[val]=true

14   Set n=(val,min_dist+1)

 15   Add n in queue

16 Procedure min_throws(ladder,snake)

17   set nodes =100

18   set edge=[]

19   Repeat steps 19 to 28 till the value of nodes

20   Set J=1

21   Repeat steps 21 to 28 till j<=6 and i+j<=nodes

22   Set source=i

23   Set lad1=ladder.get(i+j) if ladder.get(i+j) exist

24   Set snake=snake.get(i+j) if snake.get(i+j) exist

25   If lad1 or snake1 exists then set destination=lad1+snake1

26   Else destination=i+j

27   append value of source and destination in edges

28   Set j=j+1

29   Create instance of class graph g=graph()

30   Repeat procedure Breadth(g,nodes,0)

CODE

```python
from collections import deque
class Graph:
    def __init__(self, edges, nodes):
    self.ad_list = [[] for _ in range(nodes)]
    for (source, destination) in edges:
    self.ad_list[source].append(destination)
```

```python
def Breadth(graph, source, nodes):
        found = [False] * (nodes + 1)
        q1 = deque()
        found[source] = True
        q1.append((source, 0))
        while q1:
        vertex, min_dist = q1.popleft()
        if vertex == nodes:
        print("The minimum throws will be", min_dist)
        break
        for val in graph.ad_list[vertex]:
        if not found[val]:
                found[val] = True
                n = (val, min_dist + 1)
                q1.append(n)
def min_throws(ladder, snake):
        nodes = 100
        edges = []
        for i in range(nodes):
        j = 1
        while j <= 6 and i + j <= nodes:
        source = i
        lad1 = ladder.get(i + j) if (ladder.get(i + j)) else 0
        snake1 = snake.get(i + j) if (snake.get(i + j)) else 0
        if lad1 or snake1:
                destination = lad1 + snake1
        else:
                destination = i + j
        edges.append((source, destination))
        j = j + 1
        g = Graph(edges, nodes)
        Breadth(g, 0, nodes)
if __name__ == '__main__':
        snake = {}
        ladder = {}
        snake[17] = 7
        snake[54] = 34
        snake[62] = 19
        snake[64] = 60
```
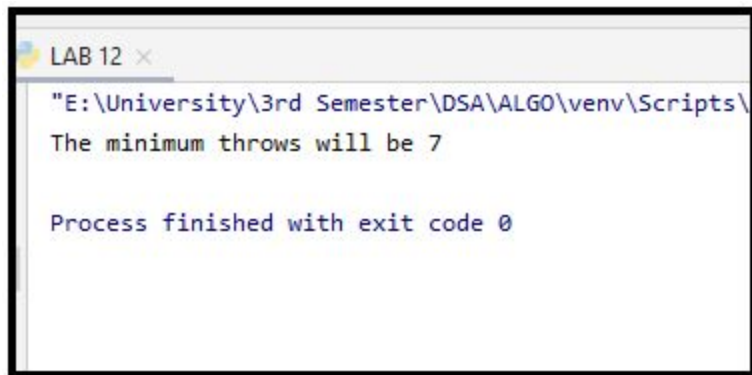
snake[87] = 36
snake[93] = 73
snake[95] = 75
snake[98] = 79
ladder[1] = 38
ladder[4] = 14
ladder[9] = 31
ladder[21] = 42
ladder[28] = 84
ladder[51] = 67
ladder[72] = 91
ladder[80] = 99
min_throws(ladder, snake)

OUTPUT

```
LAB 12 ×

"E:\University\3rd Semester\DSA\ALGO\venv\Scripts\
The minimum throws will be 7


Process finished with exit code 0
```

Lab # 13

Q # 01. What approach do you used to solve problem and why?

Ans. The depth-first search approach is used to solve the given problem because it is the best approach to find out the path between the vertices rapidly. First of all, a 2D array of maze is converted so that it is a 1D array. Now edges are created so that the rat can move from one block to another block. After it, DFS is applied to find the path. Then the path is converted into a 2D array to represent the path followed by the rat. DFS will solve this problem by taking one block then moving to its adjacent ones and repeating this process, if the destination is not found then it will backtrack and find the path from other adjacent blocks.

Q # 02. Give the algorithm of above problem, Also analyze the time & space complexity

of your algorithm.

Algorithm

1. First make a sol matrix which will show the final solution and make all its elements initially 0. 2. Now make a recursive function which will take the matrix **maze** as argument and outputs the matrix **sol** and position of rat **(i, j)**
3. Now if the position of rat is out of the matrix or position of rat is not valid then return. 4. Now mark the position of rat which is sol[i][j] as 1 and check if the current position of rat is the destination or not. If the destination is reached then print the solution matrix and return.
5. Now recursively call for position of rat as (i+1, j) and (i, j+1).
6. Then unmark the position of rat (i, j) as sol[i][j] = 0.

Time Complexity:

The time complexity of above algorithm is $O(2^{n^{\wedge}2})$

Space Complexity:

The space complexity of above algorithm is $O(n^2)$

Q # 03. Give implementation of Rat in a Maze problem.

Code:

N = 4

```python
def printSolution( sol ):

    for i in sol:
        for j in i:
            print(str(j) + " ", end ="")
        print("")

def isSafe( maze, x, y ):

    if x >= 0 and x < N and y >= 0 and y < N and maze[x][y] == 1:
        return True

    return False

def solveMaze( maze ):

    sol = [ [ 0 for j in range(4) ] for i in range(4) ]

    if solveMazeUtil(maze, 0, 0, sol)
        == False: print("Solution doesn't
        exist");
        return False

    printSolution(sol)
    return True

def solveMazeUtil(maze, x, y, sol):

    if x == N - 1 and y == N - 1 and
        maze[x][y]== 1: sol[x][y] = 1
        return True

    if isSafe(maze, x, y) == True:

        sol[x][y] = 1

        if solveMazeUtil(maze, x + 1, y, sol)
            == True: return True

        if solveMazeUtil(maze, x, y + 1, sol)
            == True: return True

        sol[x][y] = 0
        return False
```

```python
if __name__ == "__main__":

    maze = [[1, 0, 0, 0],
            [1, 1, 0, 1],
            [0, 1, 0, 0],
            [1, 1, 1, 1]]

    solveMaze(maze)
```

Output:

1 0 0 0

1 1 0 0

0 1 0 0

0 1 1 1

Lab # 14

Q # 01. What approach do you use to solve problems and why?

Answer:

The approach used in solving this problem is backtracking algorithm

Backtracking is similar to the brute force approach where it tries all of the solutions but the only difference is that it eliminates/avoids the partial candidate solutions as soon as it finds that that path cannot lead to a solution

In 4- queens problem, we have 4 queens to be placed on a 4*4 chessboard, satisfying the constraint that no two queens should be in the same row, same column, or in the same diagonal.

We can solve 4-queens problem through backtracking by taking it as a bounding function .in use the criterion that if (x1, x2, ……., xi) is a path to a current E-node, then all the children nodes with parent-child labelings x (i+1) are such that (x1, x2, x3, ….., x(i+1)) represents a chessboard configuration in which no queens are attacking.

Q # 02. Give the algorithm of the above problem , Also analyze the time & space complexity of your algorithm.

Algorithm:

SetLP(l,i){

  for j := 1 to l-1 :

    // Two in the same column// or in the same diagonal

    if ((x[j] == i) or

      (abs(x[j] – i) = Abs(j – l))):

     then return false;

  else:

     return true

KN(l,m):

   for i:= 1 to m :

      if SetLP(l, i) then

    Set x[l] = i

    if (l == m):

         print(x[1:m])

    else :

    KN(l+1, m);

Time Complexity:

Time Complexity: $O(n^2 * n!)$

We used recursion to brute-force traverse this tree, trying every possible queen combination (other than the combinations that immediately fail). We never place a queen on the same column as a previous queen, which is why the number of choices on each row is 8, then at most 7, then at most 6, ... down to at most 1.

Our Recursion Tree will have $n + 1 = 9$ levels in it

Our root will have 1 node in it.

Our next level represents row 0 (and will have 8 nodes in it)

Our next level represents row 1 (and will have $8 \times 7 = 56$ nodes in it)

Our last level represents row 7 (and will have $8 \times 7 \times 6... = 8!$ nodes in it)

Our recursion tree will have n! leaves in the tree

Therefore, there are n! solutions to generate, and each one takes $O(n^2)$ time to copy into our solutions

Space Complexity

One way to represent our space complexity is $O(n^2 * n!)$

Q # 03. Give implementation of 8 Queen Problems.

CODE:

```python
class GfG:

    def __init__(self):
        self.arr = [0] * 10

    def canPlace(self, k, i):
        for j in range(1, k):
            if (self.arr[j] == i or
                    (abs(self.arr[j] - i) == abs(j - k))):
                return False
        return True

    def display(self, n):
        for i in range(1, n + 1):
            for j in range(1, n + 1):
                if self.arr[i] != j:
                    print("\t_", end=" ")
                else:
                    print("\tQ", end=" ")
            print()

    def nQueens(self, k, n):
        for i in range(1, n + 1):
```

```python
            if self.canPlace(k, i):
                self.arr[k] = i
                if k == n:
                    self.display(n)
                else:
                    self.nQueens(k + 1, n)

if __name__ == '__main__':
    n = 8
    obj = GfG()
    obj.nQueens(1, n)
```

OUTPUT:

```
_  _  _  _  _  _  Q  _
_  _  _  _  Q  _  _  _
_  _  _  _  _  Q  _  _
_  _  Q  _  _  _  _  _
Q  _  _  _  _  _  _  _
_  _  _  _  _  _  _  Q
_  _  _  _  Q  _  _  _
_  Q  _  _  _  _  _  _

_  _  _  Q  _  _  _  _
_  _  _  _  _  _  Q  _
_  _  _  _  _  Q  _  _
_  _  _  _  _  _  Q  _
_  _  _  _  Q  _  _  _
_  _  Q  _  _  _  _  _
_  _  _  _  Q  _  _  _
Q  _  _  _  _  _  _  _
```