INDEX

Experiment - 1

Objective : Write a program in python for any searching techniques.

Code :

For binary search -

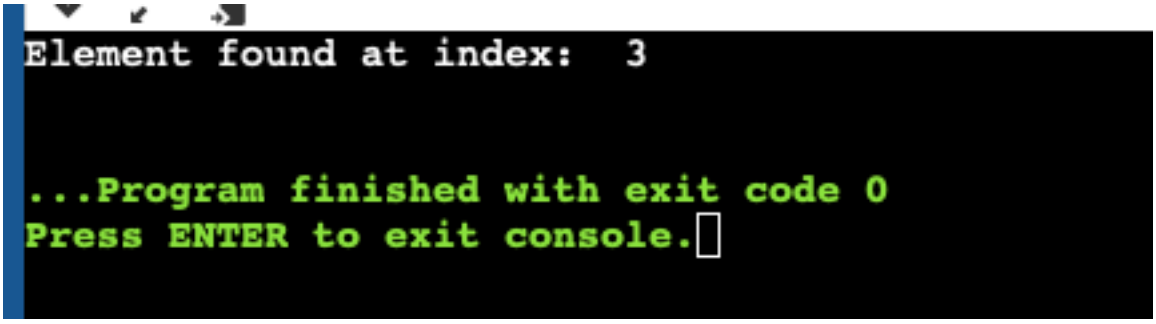
```
def binary_search(arr, low, high, a):
if high >= low:
mid = (high + low) // 2
if arr[mid] == a:
return mid
# If element is smaller than mid, then it can only present in left subarray
elif arr[mid] > a:
return binary_search(arr, low, mid - 1, a)
# Else the element can only be present in right subarray
else:
return binary_search(arr, mid + 1, high, a)
else:
# Element is not present in the array
return -1
arr = [ 20, 23, 40, 60, 90 ]
a = 23
# Function call
result = binary_search(arr, 0, len(arr)-1, a)
if result != -1:
print("Element is present at index", str(result))
else:
print("Element is not present in array")
```

Output :

For Linear Search -

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

    # Going through array sequencially
    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 :



Experiment 2

Objective :Write a program to implement BFS using python.

Code :

```
graph = {

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

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

'C' : ['F'],

'D' : [],

'E' : ['F'],

'F' : []

}


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

queue = [] #Initialize a queue


def bfs(visited, graph, node):

visited.append(node)

queue.append(node)


while queue:

s = queue.pop(0)

print (s, end = " ")


for neighbour in graph[s]:

if neighbour not in visited:

visited.append(neighbour)

queue.append(neighbour)


# Driver Code
```

bfs(visited, graph, 'A')

Output :

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL
[Running] python -u "c:\Users\hp\Desktop\python\bfs.py"
A B C D E F
[Done] exited with code=0 in 0.335 seconds
```

Experiment 3

Objective :Write a program to implement DFS using python.

Code :

```
# DFS algorithm

def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)

    print(start)

    for next in graph[start] - visited:
        dfs(graph, next, visited)
    return visited


graph = {'0': set(['1', '2']),
         '1': set(['0', '3', '4']),
         '2': set(['0']),
         '3': set(['1']),
         '4': set(['2', '3'])}

dfs(graph, '0')
```

Output :

Experiment - 4

Objective : WAP to implement TicTac Toe.

Code :

```
import os
import time

board = [' ',' ',' ',' ',' ',' ',' ',' ',' ',' ']
player = 1

########win Flags##########
Win = 1
Draw = -1
Running = 0
Stop = 1
###########################
Game = Running
Mark = 'X'

#This Function Draws Game Board
def DrawBoard():
   print(" %c | %c | %c " % (board[1],board[2],board[3]))
   print("___|___|___")
   print(" %c | %c | %c " % (board[4],board[5],board[6]))
   print("___|___|___")
   print(" %c | %c | %c " % (board[7],board[8],board[9]))
   print("   |   |   ")

#This Function Checks position is empty or not
def CheckPosition(x):
   if(board[x] == ' '):
      return True
   else:
      return False

#This Function Checks player has won or not
def CheckWin():
   global Game
   #Horizontal winning condition
   if(board[1] == board[2] and board[2] == board[3] and board[1] != ' '):
      Game = Win
   elif(board[4] == board[5] and board[5] == board[6] and board[4] != ' '):
      Game = Win
   elif(board[7] == board[8] and board[8] == board[9] and board[7] != ' '):
      Game = Win
   #Vertical Winning Condition
   elif(board[1] == board[4] and board[4] == board[7] and board[1] != ' '):
      Game = Win
   elif(board[2] == board[5] and board[5] == board[8] and board[2] != ' '):
```

```
        Game = Win
    elif(board[3] == board[6] and board[6] == board[9] and board[3] != ' '):
        Game=Win
    #Diagonal Winning Condition
    elif(board[1] == board[5] and board[5] == board[9] and board[5] != ' '):
        Game = Win
    elif(board[3] == board[5] and board[5] == board[7] and board[5] != ' '):
        Game=Win
    #Match Tie or Draw Condition
    elif(board[1]!=' ' and board[2]!=' ' and board[3]!=' ' and board[4]!=' ' and board[5]!=' ' and
board[6]!=' ' and board[7]!=' ' and board[8]!=' ' and board[9]!=' '):
        Game=Draw
    else:
        Game=Running

print("Player 1 [X] --- Player 2 [O]\n")
print()
print()
print("Please Wait...")
time.sleep(3)
while(Game == Running):
    os.system('cls')
    DrawBoard()
    if(player % 2 != 0):
        print("Player 1's chance")
        Mark = 'X'
    else:
        print("Player 2's chance")
        Mark = 'O'
    choice = int(input("Enter the position between [1-9] where you want to mark : "))
    if(CheckPosition(choice)):
        board[choice] = Mark
        player+=1
        CheckWin()

os.system('cls')
DrawBoard()
if(Game==Draw):
    print("Game Draw")
elif(Game==Win):
    player-=1
    if(player%2!=0):
        print("Player 1 Won")
    else:
        print("Player 2 Won")
```

Output :

```
Player 1 [X] --- Player 2 [O]



Please Wait...
sh: 1: cls: not found
     |     |
_____|_____|_____
     |     |
_____|_____|_____
     |     |
     |     |
Player 1's chance
Enter the position between [1-9] where you want to mark : 
```

Experiment - 5

Objective : WAP to implement heuristic search.

Code : For A* search algorithm

```
from collections import deque

class Graph:
    # example of adjacency list (or rather map)
    # adjacency_list = {
    # 'A': [('B', 1), ('C', 3), ('D', 7)],
    # 'B': [('D', 5)],
    # 'C': [('D', 12)]
    # }

    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list

    def get_neighbors(self, v):
        return self.adjacency_list[v]

    # heuristic function with equal values for all nodes
    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }

        return H[n]

    def a_star_algorithm(self, start_node, stop_node):
        # open_list is a list of nodes which have been visited, but who's neighbors
        # haven't all been inspected, starts off with the start node
        # closed_list is a list of nodes which have been visited
        # and who's neighbors have been inspected
        open_list = set([start_node])
        closed_list = set([])

        # g contains current distances from start_node to all other nodes
        # the default value (if it's not found in the map) is +infinity
        g = {}

        g[start_node] = 0

        # parents contains an adjacency map of all nodes
        parents = {}
        parents[start_node] = start_node

        while len(open_list) > 0:
```

```
n = None

# find a node with the lowest value of f() - evaluation function
for v in open_list:
    if n == None or g[v] + self.h(v) < g[n] + self.h(n):
        n = v;

if n == None:
    print('Path does not exist!')
    return None

# if the current node is the stop_node
# then we begin reconstructin the path from it to the start_node
if n == stop_node:
    reconst_path = []

    while parents[n] != n:
        reconst_path.append(n)
        n = parents[n]

    reconst_path.append(start_node)

    reconst_path.reverse()

    print('Path found: {}'.format(reconst_path))
    return reconst_path

# for all neighbors of the current node do
for (m, weight) in self.get_neighbors(n):
    # if the current node isn't in both open_list and closed_list
    # add it to open_list and note n as it's parent
    if m not in open_list and m not in closed_list:
        open_list.add(m)
        parents[m] = n
        g[m] = g[n] + weight

    # otherwise, check if it's quicker to first visit n, then m
    # and if it is, update parent data and g data
    # and if the node was in the closed_list, move it to open_list
    else:
        if g[m] > g[n] + weight:
            g[m] = g[n] + weight
            parents[m] = n

            if m in closed_list:
                closed_list.remove(m)
                open_list.add(m)

# remove n from the open_list, and add it to closed_list
# because all of his neighbors were inspected
open_list.remove(n)
closed_list.add(n)
```

```
print('Path does not exist!')
return None
```

Output :

```
adjacency_list = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('A', 'D')
```

```
Path found: ['A', 'B', 'D']
['A', 'B', 'D']
```

Experiment - 6

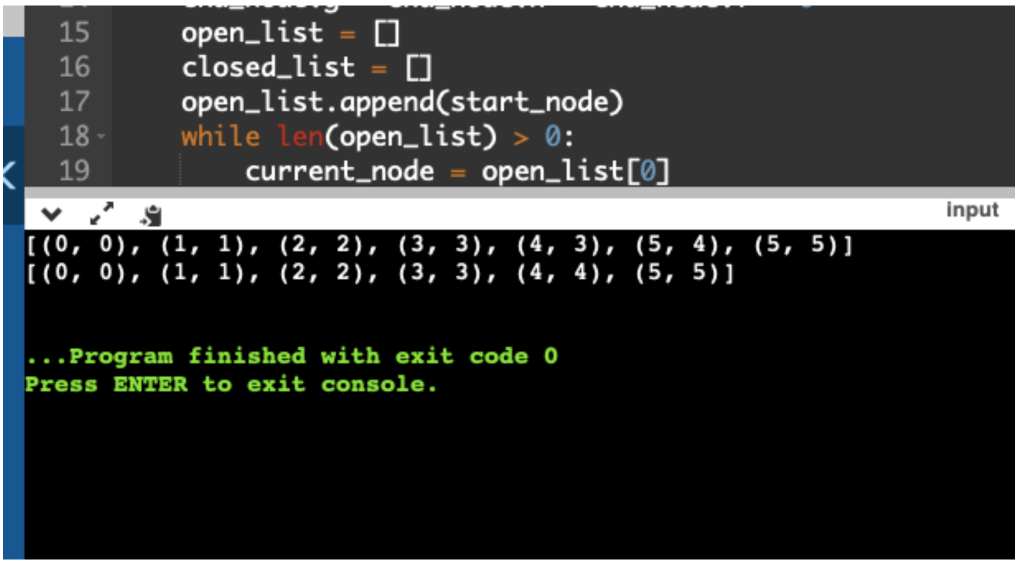
Objective : Implementation of A* algorithm.

Code :

```
class Node():

    def __init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position
        self.g = 0
        self.h = 0
        self.f = 0
    def __eq__(self, other):
        return self.position == other.position
def astar(maze, start, end):
    start_node = Node(None, start)
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, end)
    end_node.g = end_node.h = end_node.f = 0
    open_list = []
    closed_list = []
    open_list.append(start_node)
    while len(open_list) > 0:
        current_node = open_list[0]
        current_index = 0
        for index, item in enumerate(open_list):
            if item.f < current_node.f:
                current_node = item
                current_index = index
        open_list.pop(current_index)
        closed_list.append(current_node)
        if current_node == end_node:
            path = []
            current = current_node
            while current is not None:
                path.append(current.position)
                current = current.parent
            return path[::-1]
        children = []
        for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1), (1, 1)]:
            node_position = (current_node.position[0] + new_position[0], current_node.position[1] +
new_position[1])
            if node_position[0] > (len(maze) - 1) or node_position[0] < 0 or node_position[1] > (
                    len(maze[len(maze) - 1]) - 1) or node_position[1] < 0:
                continue
            if maze[node_position[0]][node_position[1]] != 0:
                continue
            new_node = Node(current_node, node_position)
            children.append(new_node)
        for child in children:
            for closed_child in closed_list:
                if child == closed_child:
                    continue
            child.g = current_node.g + 1
            child.h = ((child.position[0] - end_node.position[0]) ** 2) + (
                    (child.position[1] - end_node.position[1]) ** 2)
            child.f = child.g + child.h
```

```
            for open_node in open_list:
                if child == open_node and child.g > open_node.g:
                    continue
            open_list.append(child)
def main():
    maze = [[0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 0, 0]]
    graph = [[0, 1, 0, 0, 0, 0],
             [1, 0, 1, 0, 1, 0],
             [0, 1, 0, 0, 0, 1],
             [0, 0, 0, 0, 1, 0],
             [0, 1, 0, 1, 0, 0],
             [0, 0, 1, 0, 0, 0]
             ]
    start = (0, 0)
    end = (5, 5)
    end1 = (5, 5)
    path = astar(maze, start, end)
    print(path)
    path1 = astar(graph, start, end1)
    print(path1)
if __name__ == '__main__':
    main()
```

Output :



Experiment-7

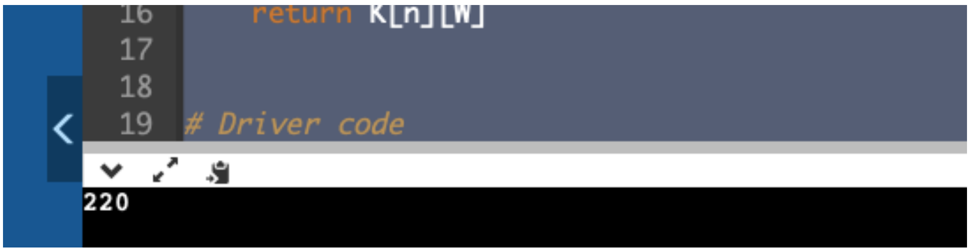
Objective : Implementation of knapsack problem.

Code :

Using dynamic approach -

```
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W + 1)] for x in range(n + 1)]

    # Build table K[][] in bottom up manner
    for i in range(n + 1):
        for w in range(W + 1):
            if i == 0 or w == 0:
                K[i][w] = 0
            elif wt[i-1] <= w:
                K[i][w] = max(val[i-1]
                          + K[i-1][w-wt[i-1]],
                            K[i-1][w])
            else:
                K[i][w] = K[i-1][w]

    return K[n][W]


# Driver code
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print(knapSack(W, wt, val, n))
```

Output :



Experiment - 8

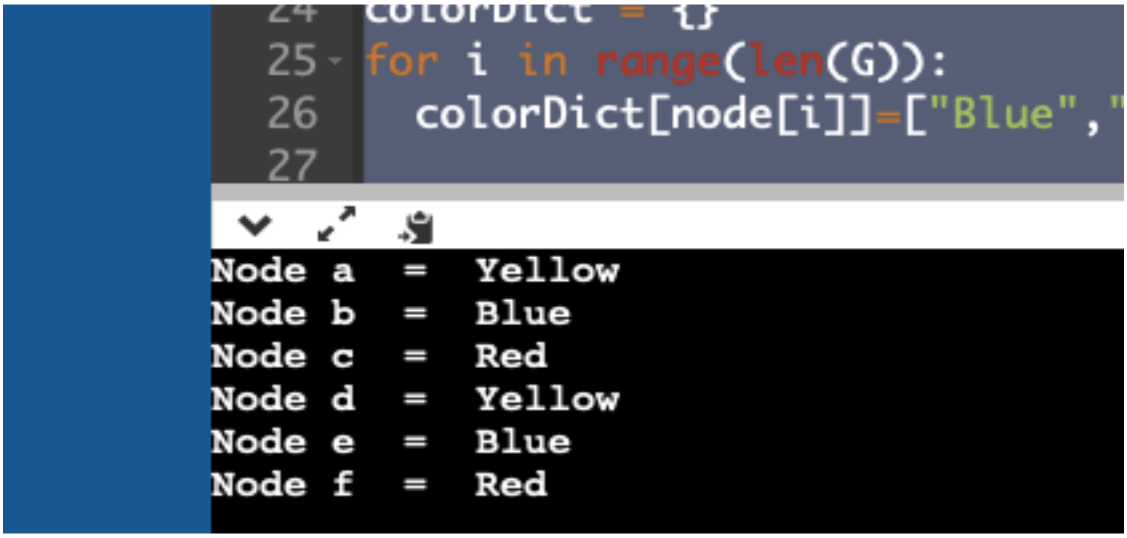
Objective : Implement Graph coloring problem using python.

Code:

```
# Adjacent Matrix
G = [[ 0, 1, 1, 0, 1, 0],
     [ 1, 0, 1, 1, 0, 1],
     [ 1, 1, 0, 1, 1, 0],
     [ 0, 1, 1, 0, 0, 1],
     [ 1, 0, 1, 0, 0, 1],
     [ 0, 1, 0, 1, 1, 0]]



# inisiate the name of node.
node = "abcdef"
t_={}
for i in range(len(G)):
  t_[node[i]] = i

# count degree of all node.
degree =[]
for i in range(len(G)):
  degree.append(sum(G[i]))

# inisiate the posible color
colorDict = {}
for i in range(len(G)):
  colorDict[node[i]]=["Blue","Red","Yellow","Green"]


# sort the node depends on the degree
sortedNode=[]
indeks = []

# use selection sort
for i in range(len(degree)):
  _max = 0
  j = 0
  for j in range(len(degree)):
    if j not in indeks:
      if degree[j] > _max:
        _max = degree[j]
        idx = j
  indeks.append(idx)
  sortedNode.append(node[idx])

# The main process
theSolution={}
for n in sortedNode:
  setTheColor = colorDict[n]
  theSolution[n] = setTheColor[0]
  adjacentNode = G[t_[n]]
  for j in range(len(adjacentNode)):
    if adjacentNode[j]==1 and (setTheColor[0] in colorDict[node[j]]):
      colorDict[node[j]].remove(setTheColor[0])
```

```
# Print the solution
for t,w in sorted(theSolution.items()):
  print("Node",t," = “,w)
```

Output :



Experiment - 9

Objective : Tokenization of word and Sentences with the help of NLTK package.

Sentence tokenizer :

```
from nltk.tokenize import sent_tokenize
text = "Hello everyone. Welcome to NLP and the NLTK module introduction"
sent_tokenize(text)
```

Output :

```
['Hello everyone. Welcome to NLP and the NLTK module introduction']
```

Word tokenizer :

```
rom nltk.tokenize import word_tokenize
text = "Hello everyone. Welcome to NLP and the NLTK module introduction"
word_tokenize(text)
```

Output :

```
['Hello', 'everyone.', 'Welcome', 'to', 'NLP', 'and', 'the', 'NLTK', 'module', '
introduction']
```