Cse-334

Artificial intelligence Lab

Submitted To

Mosaddek Adnan Sikdar
Lecturer, Dept. of CSE
Primeasia University

Submitted By

Md.soykot
181-041-042

1. Fibonacci sequence without recursion

```
n=int(input("Enter number:"))
fiboPrev=0
fibo=1
print("Fibonacci sequence for {0} number:".format(n))

for i in range(n):
   print(fiboPrev,end=" ")
   fiboNext=fiboPrev+fibo
   fiboPrev=fibo
   fibo=fiboNext
```

2. Fibonacci Sequence Using Recursion

```
def fibo(n):
   if n<=1:
      return n
   else:
      return fibo(n-1)+fibo(n-2)

n=int(input("Enter number:"))
print("Fibonacci sequence for {0} number:".format(n))
for i in range(n):
   print(fibo(i),end=" ")
```

3. Armstrong Number in an Interval

```
print("Enter two number lower and upper range:")
lower=int(input())
```

```
upper=int(input())

for num in range(lower,upper+1):
   total=0
   temp=num
   while temp>0:
      singleDigit=temp%10
      total +=singleDigit**3
      temp //=10 #it's give only integer value
   if num==total:
       print(num)
```

4. Convert Binary to Decimal Using Recursion.

```
def  binToDec(num):
   if num ==1 or num == 0:
      return num

   length=len(str(num))
   firstDigit=num//pow(10,length-1)
   return (pow(2,length-1)*firstDigit)+binToDec(num%pow(10,length-1))

binary=int(input('Enter a binary number: '))
decimal=binToDec(binary)

print('Deccimal of {0} is {1}'.format(binary,decimal))
```

5. Convert Decimal to Binary, Octal and Hexadecimal.

```
decimal=int(input('Enter a decimal number: '))
```

```
print(decimal, "in binary",bin(decimal).replace("0b", "") )
print(decimal, "in Octal : ", oct(decimal).replace("0o", ""))
print(decimal, " in Hexadecimal : ", hex(decimal).replace("0x", ""))
```

6. Multiply Two Matrices.

```
def matrics(r,c):

    matrix=[]

    for i in range(r):
        a=[]
        for j in range(c):
            a.append(int(input()))
        matrix.append(a)
    return matrix

print("=========Enter first matrix======== ")
r1 = int(input("Enter the number of rows:"))
c1 = int(input("Enter the number of columns:"))
firstMatrix=matrics(r1,c1)

print("First matrix:")
for i in range(r1):
    for j in range(c1):
        print(firstMatrix[i][j],end=" ")
    print()

print("==========Enter second matrix========= ")
r2 = int(input("Enter the number of rows:"))
```

```
c2 = int(input("Enter the number of columns:"))
secondMatrix=matrics(r2,c2)

print("Second matrix:")
for i in range(r2):
   for j in range(c2):
      print(secondMatrix[i][j],end=" ")
   print()

result=[[0 for x in range(r1)] for y in range(c2)]

print("Multiplication matrix:")
if r1==c2:
   for i in range(len(firstMatrix)):
      for j in range(len(secondMatrix[0])):
         for k in range(len(secondMatrix)):
            result[i][j] +=firstMatrix[i][k]*secondMatrix[k][j]

   for r in result:
      print(" ".join(map(str,r)))
else:
   print("Invalid input")
```

7. Transpose a Matrix.

```
def matrics(r,c):
```

```
    matrix=[]

    for i in range(r):
        a=[]
        for j in range(c):
            a.append(int(input()))
        matrix.append(a)
    return matrix

r = int(input("Enter the number of rows:"))
c = int(input("Enter the number of columns:"))
firstMatrix=matrics(r,c)

print("matrix:")
for i in range(r):
    for j in range(c):
        print(firstMatrix[i][j],end=" ")
    print()

result=[[ 0 for x in range(r)] for y in range(c)]

for i in range(len(firstMatrix)):
        for j in range(len(firstMatrix[0])):
            result[j][i]=firstMatrix[i][j]

print("Transpose matrix:")
for r in result:
    print(r)
```

8. Count the Number of Each Vowel on a given sentence

```
sen=input("Enter a sentence: ")
lowerCase=sen.lower()
vowelCount={}
for vowel in "aeiou":
    count=lowerCase.count(vowel)
    vowelCount[vowel]=count

print(vowelCount)
```

9. A* search algorithm

```
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)]:
# Adjacent squares
```

```
        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],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

    start = (0, 0)
    end = (7, 6)

    path = astar(maze, start, end)
    print(path)


if __name__ == '__main__':
    main()
```

10. Tic-tac-toe

```
theBoard = {'7': ' ' , '8': ' ' , '9': ' ' ,
                    '4': ' ' , '5': ' ' , '6': ' ' ,
                    '1': ' ' , '2': ' ' , '3': ' ' }
```

```
board_keys = []

for key in theBoard:
   board_keys.append(key)



def printBoard(board):
   print(board['7'] + '|' + board['8'] + '|' + board['9'])
   print('-+-+-')
   print(board['4'] + '|' + board['5'] + '|' + board['6'])
   print('-+-+-')
   print(board['1'] + '|' + board['2'] + '|' + board['3'])



def game():

   turn = 'X'
   count = 0



   for i in range(10):
      printBoard(theBoard)
      print("It's your turn," + turn + ".Move to which place?")

      move = input()

      if theBoard[move] == ' ':
         theBoard[move] = turn
         count += 1
```

```
else:
    print("That place is already filled.\nMove to which place?")
    continue

if count >= 5:
    if theBoard['7'] == theBoard['8'] == theBoard['9'] != ' ':
        printBoard(theBoard)
        print("\nGame Over.\n")
        print(" **** " +turn + " won. ****")
        break
    elif theBoard['4'] == theBoard['5'] == theBoard['6'] != ' ':
        printBoard(theBoard)
        print("\nGame Over.\n")
        print(" **** " +turn + " won. ****")
        break
    elif theBoard['1'] == theBoard['2'] == theBoard['3'] != ' ':
        printBoard(theBoard)
        print("\nGame Over.\n")
        print(" **** " +turn + " won. ****")
        break
    elif theBoard['1'] == theBoard['4'] == theBoard['7'] != ' ':
        printBoard(theBoard)
        print("\nGame Over.\n")
        print(" **** " +turn + " won. ****")
        break
    elif theBoard['2'] == theBoard['5'] == theBoard['8'] != ' ':
        printBoard(theBoard)
        print("\nGame Over.\n")
        print(" **** " +turn + " won. ****")
        break
```

```
        elif theBoard['3'] == theBoard['6'] == theBoard['9'] != ' ':
            printBoard(theBoard)
            print("\nGame Over.\n")
            print(" **** " +turn + " won. ****")
            break
        elif theBoard['7'] == theBoard['5'] == theBoard['3'] != ' ':
            printBoard(theBoard)
            print("\nGame Over.\n")
            print(" **** " +turn + " won. ****")
            break
        elif theBoard['1'] == theBoard['5'] == theBoard['9'] != ' ':
            printBoard(theBoard)
            print("\nGame Over.\n")
            print(" **** " +turn + " won. ****")
            break


    if count == 9:
        print("\nGame Over.\n")
        print("It's a Tie!!")

    if turn =='X':
        turn = 'O'
    else:
        turn = 'X'

restart = input("Do want to play Again?(y/n)")
if restart == "y" or restart == "Y":
    for key in board_keys:
        theBoard[key] = " "
```

```
        game()

if __name__ == "__main__":
    game()
```

11. BFS Implementation

```
graph = {
    1:[2,3,4],
    2:[],
    3:[5],
    4:[6],
    5:[],
    6:[]

}
visited = []
queue = []

def bfs( node):
    visited.append(node)
    queue.append(node)

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

        for neighbour in graph:
            if neighbour not in visited:
                visited.append(neighbour)
```

```
            queue.append(neighbour)

bfs(1)
```

12. DFS Implementation

```
graph = {
   1:[2,3],
   2:[4,5],
   3:[6],
   4:[],
   5:[6],
   6:[]

}

visited =[]

print("Depth first search:")
def dfs(node):
   if node not in visited:
      print (node,end=" ")
      visited.append(node)
      for neighbour in graph[node]:
         dfs(neighbour)

dfs(1)
```

13. Depth Limited Search Implementation

```
graph = {
   1:[2,3],
   2:[4,5],
   3:[6],
   4:[],
   5:[6],
   6:[]

}

visited =[]

print("Depth limited search:")
def dfs(node,limit):
   if not limit:
      return 0
   else:
      limit -=1
   if node not in visited:
      print (node,end=" ")
      visited.append(node)
      for neighbour in graph[node]:
         dfs(neighbour,limit)

dfs(1,2)
```

14. Naive Bayes (Probability of Playing if Raining)

```
weather_tbl=[]
```

```
play_tbl=[]

n=int(input("Enter number of dataset:"))

for i in range(n):
    weather=input("Enter weather name:")
    weather_tbl.append(weather.lower())
    play=input("Enter paly or not:")
    play_tbl.append(play.lower())



t_rainy_y,t_rainy_n=0,0
t_y,t_n=0,0
t_rainy=0

for i in range(n):
    if play_tbl[i]=="yes":
        t_y +=1
    else:
        t_n +=1
    if weather_tbl[i]=="rainy":
        t_rainy +=1
        if play_tbl[i]=="yes":
            t_rainy_y +=1
        else:
            t_rainy_n +=1

p_rainy_y=(t_rainy_y/t_y)
p_y=(t_y/n)
```

```
p_rainy=(t_rainy/n)
p_rainy_n=(t_rainy_n/t_n)
p_n=(t_n/n)

p_y_rainy=(p_rainy_y*p_y)/p_rainy
p_n_rainy=(p_rainy_n*p_n)/p_rainy

if p_n_rainy>p_y_rainy:
   print("Player will play,if it not rainy")
else:
   print("player will not play if it rainy")
```

15. Naive Bayes (Classifying Unknown Fruit)

```
typ_col=[]
lon_col=[]
n_lon_col=[]
swt_col=[]
n_swt_col=[]
yell_col=[]
n_yell_col=[]
total_col=[]

d_set=int(input("Enter number of dataset:"))
n=int(input("Eneter total fruits:"))

for i in range(n):
   typ=input("Enter fruits type:")
   typ_col.append(typ.lower())
   lon=int(input("Enter long fruits:"))
```

```
    lon_col.append(lon)
    n_lon=int(input("Enter not long fruits:"))
    n_lon_col.append(n_lon)
    swt=int(input("Enter sweet fruits:"))
    swt_col.append(swt)
    n_swt=int(input("Enter not sweet fruits:"))
    n_swt_col.append(n_swt)
    yell=int(input("Enter yellow fruits:"))
    yell_col.append(yell)
    n_yell=int(input("Enter not yellow fruits:"))
    n_yell_col.append(n_yell)
    total=int(input("Enter total fruits:"))
    total_col.append(total)

p_banana,p_orange,p_other=0,0,0

for i in range(n):
    if typ_col[i]=="banana":
        p_banana=total_col[i]/d_set
        p_long_b=lon_col[i]/total_col[i]
        p_sweet_b=swt_col[i]/total_col[i]
        p_yell_b=yell_col[i]/total_col[i]

    elif typ_col[i]=="orange":
        p_orange=total_col[i]/d_set
        p_long_o=lon_col[i]/total_col[i]
        p_sweet_o=swt_col[i]/total_col[i]
        p_yell_o=yell_col[i]/total_col[i]

    elif typ_col[i]=="other":
```

```
        p_other=total_col[i]/d_set
        p_long_other=lon_col[i]/total_col[i]
        p_sweet_other=swt_col[i]/total_col[i]
        p_yell_other=yell_col[i]/total_col[i]

p_fruits_b=p_long_b*p_sweet_b*p_yell_b
p_fruits_o=p_long_o*p_sweet_o*p_yell_o
p_fruits_other=p_long_other*p_sweet_other*p_yell_other

new_fruit=max(p_fruits_b,p_fruits_o,p_fruits_other)

if new_fruit==p_fruits_b:
   print("========New fruit will be banana==========")
elif new_fruit==p_fruits_o:
   print("==========New fruit will be orange=======")
else:
   print("======new fruits will be other fruits=======")
```