**Lab Task 1**

Triangle Draw in python:

num\_rows = int(input("Enter the number of rows: "));

k = 1

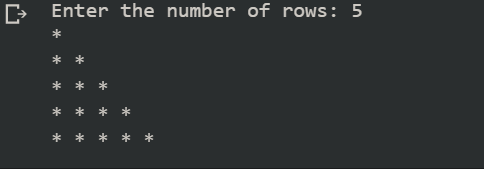
for i in range(0, num\_rows):

    for j in range(0, k):

        print("\* ", end="")

    k = k + 1

    print()



Calculator:

num1 = int(input("Enter 1st number: "))

num2 = int(input("Enter 2nd number: "))

print("1-Sum  2-Product  3-Subtraction 4-Division  5-Modulus  6-Power of num1 with num2   7-Floor division")

n=int(input("Select respective number for operation from above list: "))

if(n==1):

  print("Sum of %d and %d is: " % (num1, num2),num1+num2)

elif(n==2):

  print("Product of %d and %d is: "%(num1,num2),num1\*num2)

elif(n==3):

  print("Subtraction of %d and %d is: " %(num1,num2),num1-num2)

elif(n==4):

  print("Division of %d and %d is: " %(num1,num2),num1/num2)

elif(n==5):

  print("Modulus of %d and %d is: " %(num1,num2),num1%num2)

elif(n==6):

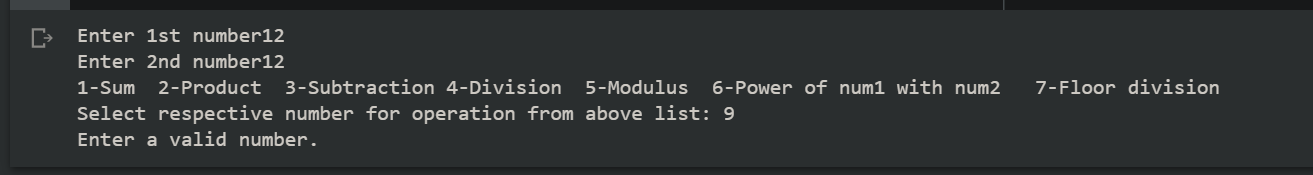
  print("Power of %d with %d is: " %(num1,num2),num1\*\*num2)

elif(n==7):

  print("Floor division of %d with %d is: " %(num1,num2),num1//num2)

else:

  print("Enter a valid number.")



Fibonacci:

n\_terms = int(input("How many terms you wants to print? "))

f1 = 0

f2 = 1

count = 0

if n\_terms <= 0:

  print ("Please enter a positive integer.")

elif n\_terms == 1:

    print ("The Fibonacci sequence up to", n\_terms, ": ")

    print(f1)

else:

    print ("The fibonacci sequence is: ")

    while count < n\_terms:

        print(f1)

        nth = f1 + f2

        f1 = f2

        f2 = nth

        count += 1

**Lab Task 2**

**Dictionary:**

#dictionary in python

countryCity = {"Pakistan": "Karachi","Turkey": "Istanbul", "Japan": "Tokyo","England": "London"}

print(countryCity)

#inserting element in dictionary

countryCity["Sri Lanka"] = "Colombo"

#updating dictionary

countryCity["Pakistan"] = "Islamabad"

print(countryCity)

#deleting element from dictionary

del countryCity["Japan"]

print(countryCity)

#sorting dictionary elements

sortedList = sorted(countryCity)

print("sorted list is")

print(sortedList)

#getting all only keys of dictionaries

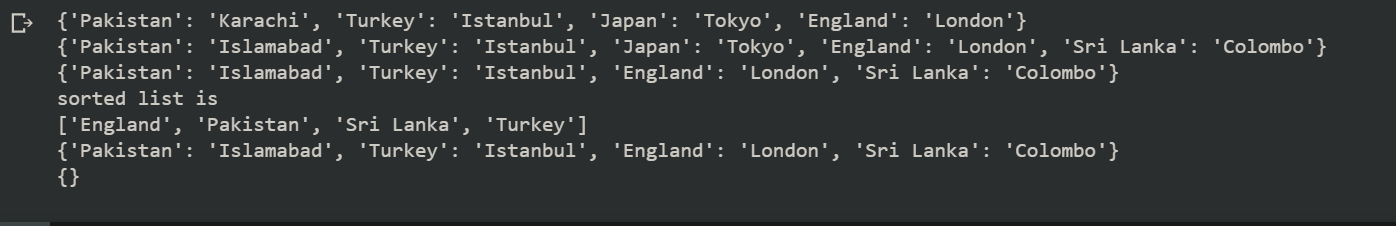
countryCity.keys()

print(countryCity)

#deleting whole dictionary

countryCity.clear();

print(countryCity)



**Taking input in data structures and printing them:**

#implementation of function in python and printing tuple, set, dictionary, list by taking input from users

print("1-Insert in List   2-Insert in Set   3-Insert in Tuple    4-Insert in Dictionary")

choice = int(input("Enter your choice from above: "))

numberOfElements = int(input("Enter number of elements you want to insert: "))

List = ["Apple"]

Tuple = ()

Set = {}

Dictionary = {}

if(choice==1):

  for x in range(numberOfElements):

    element = input("Enter element: ")

    List.append(element)

  print(List)

elif(choice==2):

  for x in range(numberOfElements):

    element = input("Enter element: ")

    Set.add(element)

  print(Set)

elif(choice==3):

  for x in range(numberOfElements):

    element = input("Enter element: ")

    Tuple = Tuple + (element,)

  print(Tuple)

elif(choice==4):

  for x in range(numberOfElements):

    element = input("Enter element: ")

    Dictionary[x] = element

  print(Dictionary)

def printDataStructures():

  if(choice==1):

    print("List is: ", List)

  elif(choice==2):

    print("Set is: " , Set)

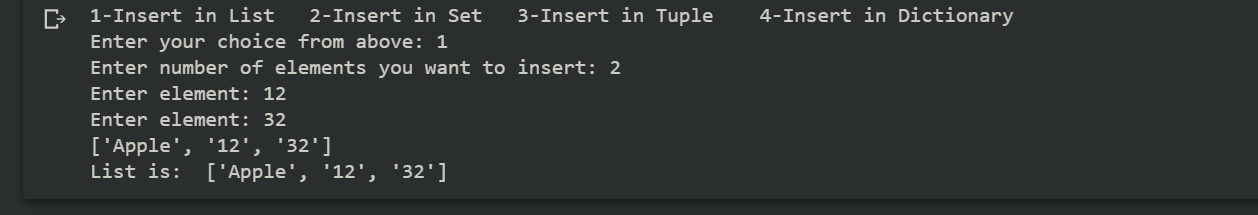
  elif(choice==3):

    print("Tuple is: " , Tuple)

  elif(choice==4):

    print("Dictionary is: " , Dictionary)

printDataStructures()



**CRUD operations on All Data Structures:**

**Set:**

#CRUD on set

Set = {0,12,3}

print(Set)

print("1-Insert   2-Delete   3-Update   4-Read")

choice = int(input("Enter your choice: "))

if(choice==1):

  element = int(input("Enter element: "))

  Set.add(element)

  print(Set)

elif(choice==2):

  element = int(input("Enter element you want to delete: "))

  for element in Set:

    if(element):

      Set.discard(element)

  print(Set)

elif(choice==3):

  element = int(input("Enter element you want to update: "))

  new = int(input("Enter new element: "))

  Set.remove(element)

  Set.add(new)

  print(Set)

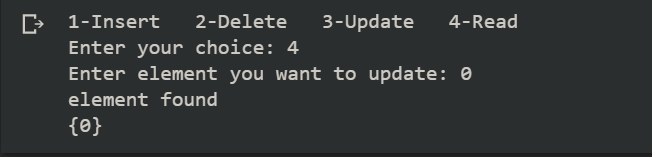
elif(choice==4):

  element = int(input("Enter element you want to update: "))

  for element in Set:

    print("element found")

  print(Set)



**List:**

#CRUD on List

List = ['ahmad','rehan']

print(List)

print("1-Insert   2-Delete   3-Update   4-Read")

choice = int(input("Enter your choice: "))

if(choice==1):

  element = input("Enter element: ")

  List.append(element)

  print(List)

elif(choice==2):

  element = input("Enter element you want to delete: ")

  List.remove(element)

  print(List)

elif(choice==3):

  element = input("Enter element you want to update: ")

  index = 0

  if element in List:

    index = List.index(element)

  new = input("Enter new element: ")

  List[index] = new

  print(List)

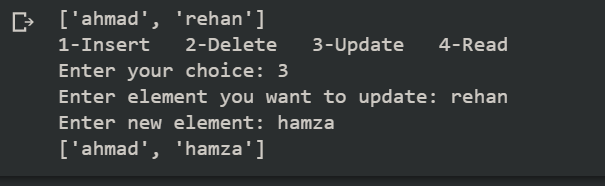
elif(choice==4):

  element = input("Enter element you want to update: ")

  if element in List:

    print("element found")

  print(List)



**Dictionary**:

#CRUD on Dictionary

Dictionary = {"England": "London"}

print(Dictionary)

print("1-Insert   2-Delete   3-Update   4-Read")

choice = int(input("Enter your choice: "))

if(choice==1):

  key = input("Enter key: ")

  value = input("Enter value of key: ")

  Dictionary[key] = value

  print(Dictionary)

elif(choice==2):

  key = input("Enter key you want to delete: ")

  del Dictionary[key]

  print(Dictionary)

elif(choice==3):

  key = input("Enter key whose value you want to update: ")

  value = input("Enter new value: ")

  Dictionary[key] = value

  print(Dictionary)

elif(choice==4):

  element = input("Enter key you want to search: ")

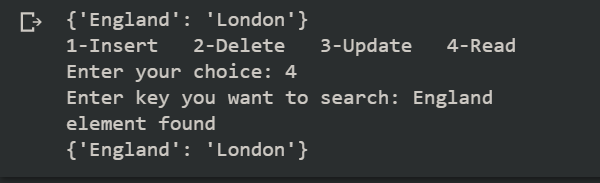
  if element in Dictionary:

    print("Element found")

  else:

    print("Element not found - check spellings")

  print(Dictionary)



**Tuple**:

**Merging in all Data Structures:**

#merging sets

Set1 = {1,2,3}

Set2 = {4,5,6}

Set3 = Set1.union(Set2)

print(Set3)

#merging Tuples

T1 = (1,2,3)

T2 = (4,5,6)

T3 = T1 + T2

print(T3)

#merging Lists

L1 = [1,2,3]

L2 = [4,5,6]

L1.extend(L2)

print(L1)

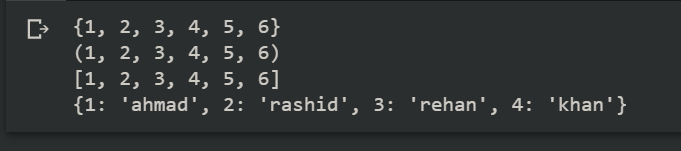
#merging Dictionaries

D1 = {1:'ahmad',2:'rashid'}

D2 = {3:'rehan',4:'khan'}

D1.update(D2)

print(D1)



**Lab Task 3**

class Library:

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

self.books = {'Intro to CS': {'Book Name': 'Intro to CS', 'Author Name': 'Ahmad', 'Book ID': '123', 'No of Copies': 3}}

def addBook(self):

name = input("Enter the name of book: ")

author = input("Enter the author of book: ")

id = input("Enter the ISBN of book: ")

copies = int(input('Enter the number of copies: '))

self.books[name] = {'Book Name': name, 'Author Name': author, 'Book ID': id, 'No of Copies': copies}

def removeBook(self):

name = input("Enter the name of book you want to delete: ")

del self.books[name]

def searchBook(self):

name = input("Enter the name of book you want to search: ")

if self.books.get(name):

print('Book found.')

print(self.books[name])

def updateBook(self):

name = input("Enter the name of book you want to update: ")

author = input("Enter the author of book: ")

id = input("Enter the ISBN of book: ")

self.books[name] = {'Book Name': name, 'Author Name': author, 'Book ID': id, 'No of Copies': self.books[name]['No of Copies']}

def viewBooks(self):

print(self.books)

def lendBook(self):

name = input("Enter the name of book you want to lend: ")

print(self.books[name])

if self.books.get(name):

copy = self.books[name]['No of Copies']

copy = copy - 1

self.books[name]['No of Copies'] = copy

print(self.books[name])

def returnBook(self):

name = input("Enter the name of book you want to return: ")

print(self.books[name])

if self.books.get(name):

copy = self.books[name]['No of Copies']

copy = copy + 1

self.books[name]['No of Copies'] = copy

print(self.books[name])

library = Library()

while True:

print('1-Add book 2-Delete book 3-Update book 4-Search book \n5-View all books 6-Lend book 7-Return book')

choice = int(input("Enter your choice: "))

if choice == 1:

library.addBook()

print(books)

elif choice == 2:

library.removeBook()

print(books)

elif choice == 3:

library.updateBook()

print(books)

elif choice == 4:

library.searchBook()

elif choice == 5:

library.viewBooks()

elif choice == 6:

library.lendBook()

print(books)

elif choice == 7:

library.returnBook()

print(books)

else:

break

**Lab Task 4**

graph = {'a':['b','c'],'b':['a','d'],'c':['a','d'],'d':['b','c','e'],'e':['d']}

print(graph)

print('Vertices of graph are: ',list(graph.keys()))

print('Edges of graph are: ', list(graph.values()))

#adding a new vertex

def addVertex():

    vertex = input('Enter vertex: ')

    if(vertex in graph):

        while True:

            edge = input('Enter edge: ')

            graph[vertex].append(edge)

            if(edge in graph):

                    print(edge)

                    graph[edge].append(vertex)

            more = input('Do you want to add more edges(y/n): ')

            if(more=='n'):

                break

    else:

        graph[vertex] = []

        while True:

            edge = input('Enter edge: ')

            graph[vertex].append(edge)

            if(edge in graph):

                print(edge)

                graph[edge].append(vertex)

            more = input('Do you want to add more edges(y/n): ')

            if(more=='n'):

                break

    print(graph)

#DFS traversal

def DFS(graph, start, visited=None):

    if visited is None:

        visited = set()

    visited.add(start)

    print(str(start) + " ", end="")

    for x in graph[start]:

        if x not in visited:

            DFS(graph, x, visited)

#BFS traversal

from collections import deque

def BFS(graph, start):

    visited = set()

    queue = deque([start])

    visited.add(start)

    while queue:

        vertex = queue.popleft()

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

        for neighbor in graph[vertex]:

            if neighbor not in visited:

                visited.add(neighbor)

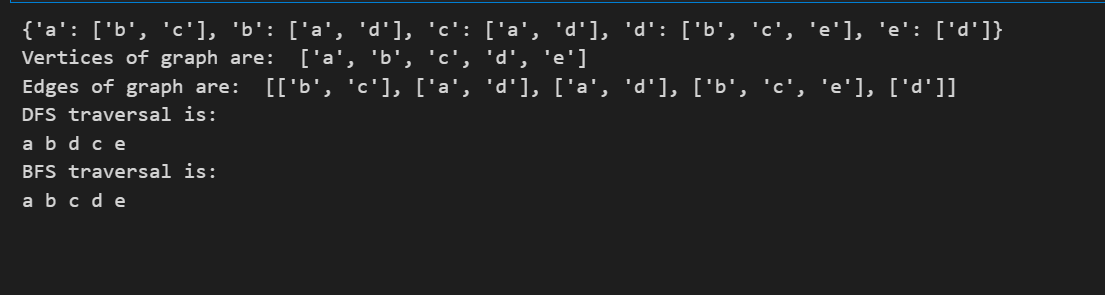
                queue.append(neighbor)

print('DFS traversal is: ')

DFS(graph,'a')

print('\nBFS traversal is: ')

BFS(graph,'a')



**Task 5**

graph = {

    'Oradea':{'Zerind':71,'Sibiu':151},

    'Zerind':{'Arad':75,'Oradea':71},

    'Arad':{'Timisoara':118,'Sibiu':140,'Zerind':75},

    'Sibiu':{'Fagaraw':99,'Rimnicu Vilcea':80,'Oradea':151,'Arad':140},

    'Timisoara':{'Lugoj':111,'Arad':118},

    'Lugoj':{'Timisoara':111,'Mehadia':70},

    'Rimnicu Vilcea':{'Sibiu':80,'Craiova':146,'Pitesti':97},

    'Mehadia':{'Lugoj':70,'Drobeta':75},

    'Drobeta':{'Mehadia':75,'Craiova':120},

    'Pitesti':{'Rimnicu Vilcea':97,'Craiova':138,'Bucharest':101},

    'Craiova':{'Drobeta':120,'Rimnicu Vilcea':146,'Pitesti':138},

    'Fagaraw':{'Sibiu':99,'Bucharest':211},

    'Bucharest':{'Pitesti':101,'Giurgin':90,'Fagaraw':211,'Urziceni':85},

    'Giurgin':{'Bucharest':90},

    'Urziceni':{'Bucharest':85,'Hirsova':98,'Vaslui':142},

    'Eforie':{'Hirsova':86},

    'Hirsova':{'Eforie':86,'Urziceni':98},

    'Vaslui':{'Urziceni':142,'Iasi':92},

    'Iasi':{'Vaslui':92,'Neamt':87},

    'Neamt':{'Iasi':87}

}

import sys

def dijkstra(graph, start):

    distances = {node: sys.maxsize for node in graph}

    distances[start] = 0

    visited = set()

    while visited != graph.keys():

        node = min((set(distances.keys()) - visited), key=distances.get)

        visited.add(node)

        for neighbor, weight in graph[node].items():

            total\_distance = distances[node] + weight

            if total\_distance < distances[neighbor]:

                distances[neighbor] = total\_distance

    return distances

dijkstra(graph=graph,start='Oradea')

import heapq

def ucs(graph, start, goal):

    frontier = [(0, start)]

    explored = set()

    while frontier:

        current\_cost, current\_node = heapq.heappop(frontier)

        if current\_node == goal:

            return current\_cost

        explored.add(current\_node)

        for neighbor, cost in graph[current\_node].items():

            new\_cost = current\_cost + cost

            if neighbor not in explored:

                frontier\_cost = next((c for (c, n) in frontier if n == neighbor), None)

                if frontier\_cost is None or new\_cost < frontier\_cost:

                    heapq.heappush(frontier, (new\_cost, neighbor))

    return None

ucs(graph=graph,start='Oradea',goal='Fagaraw')



graph = {

    'Oradea':{'Zerind':71,'Sibiu':151},

    'Zerind':{'Arad':75,'Oradea':-71},

    'Arad':{'Timisoara':-118,'Sibiu':-140,'Zerind':75},

    'Sibiu':{'Fagaraw':99,'Rimnicu Vilcea':-80,'Oradea':-151,'Arad':140},

    'Timisoara':{'Lugoj':111,'Arad':118},

    'Lugoj':{'Timisoara':111,'Mehadia':70},

    'Rimnicu Vilcea':{'Sibiu':-80,'Craiova':-146,'Pitesti':-97},

    'Mehadia':{'Lugoj':70,'Drobeta':75},

    'Drobeta':{'Mehadia':-75,'Craiova':120},

    'Pitesti':{'Rimnicu Vilcea':97,'Craiova':-138,'Bucharest':-101},

    'Craiova':{'Drobeta':120,'Rimnicu Vilcea':146,'Pitesti':138},

    'Fagaraw':{'Sibiu':99,'Bucharest':211},

    'Bucharest':{'Pitesti':-101,'Giurgin':90,'Fagaraw':-211,'Urziceni':-85},

    'Giurgin':{'Bucharest':90},

    'Urziceni':{'Bucharest':85,'Hirsova':98,'Vaslui':142},

    'Eforie':{'Hirsova':-86},

    'Hirsova':{'Eforie':86,'Urziceni':98},

    'Vaslui':{'Urziceni':-142,'Iasi':-92},

    'Iasi':{'Vaslui':92,'Neamt':87},

    'Neamt':{'Iasi':-87}

}

import heapq

def ucs(graph, start, goal):

    frontier = [(0, start)]

    explored = set()

    while frontier:

        current\_cost, current\_node = heapq.heappop(frontier)

        if current\_node == goal:

            return current\_cost

        explored.add(current\_node)

        for neighbor, cost in graph[current\_node].items():

            new\_cost = current\_cost + cost

            if neighbor not in explored:

                frontier\_cost = next((c for (c, n) in frontier if n == neighbor), None)

                if frontier\_cost is None or new\_cost < frontier\_cost:

                    heapq.heappush(frontier, (new\_cost, neighbor))

    return None

ucs(graph=graph,start='Drobeta',goal='Iasi')



graph = {

    'Oradea':{'Zerind':-71,'Sibiu':151},

    'Zerind':{'Arad':-75,'Oradea':71},

    'Arad':{'Timisoara':-118,'Sibiu':140,'Zerind':75},

    'Sibiu':{'Fagaraw':-99,'Rimnicu Vilcea':80,'Oradea':151,'Arad':140},

    'Timisoara':{'Lugoj':-111,'Arad':118},

    'Lugoj':{'Timisoara':-111,'Mehadia':70},

    'Rimnicu Vilcea':{'Sibiu':-80,'Craiova':146,'Pitesti':97},

    'Mehadia':{'Lugoj':-70,'Drobeta':75},

    'Drobeta':{'Mehadia':-75,'Craiova':120},

    'Pitesti':{'Rimnicu Vilcea':-97,'Craiova':138,'Bucharest':101},

    'Craiova':{'Drobeta':-120,'Rimnicu Vilcea':146,'Pitesti':138},

    'Fagaraw':{'Sibiu':-99,'Bucharest':211},

    'Bucharest':{'Pitesti':-101,'Giurgin':90,'Fagaraw':211,'Urziceni':85},

    'Giurgin':{'Bucharest':-90},

    'Urziceni':{'Bucharest':-85,'Hirsova':98,'Vaslui':142},

    'Eforie':{'Hirsova':-86},

    'Hirsova':{'Eforie':-86,'Urziceni':98},

    'Vaslui':{'Urziceni':-142,'Iasi':92},

    'Iasi':{'Vaslui':-92,'Neamt':87},

    'Neamt':{'Iasi':-87}

}

import heapq

def ucs(graph, start, goal):

    frontier = [(0, start)]

    explored = set()

    while frontier:

        current\_cost, current\_node = heapq.heappop(frontier)

        if current\_node == goal:

            return current\_cost

        explored.add(current\_node)

        for neighbor, cost in graph[current\_node].items():

            new\_cost = current\_cost + cost

            if neighbor not in explored:

                frontier\_cost = next((c for (c, n) in frontier if n == neighbor), None)

                if frontier\_cost is None or new\_cost < frontier\_cost:

                    heapq.heappush(frontier, (new\_cost, neighbor))

    return None

ucs(graph=graph,start='Fagaraw',goal='Arad')



graph = {

    'Oradea':{'Zerind':-71,'Sibiu':-151},

    'Zerind':{'Arad':-75,'Oradea':-71},

    'Arad':{'Timisoara':-118,'Sibiu':-140,'Zerind':-75},

    'Sibiu':{'Fagaraw':-99,'Rimnicu Vilcea':-80,'Oradea':-151,'Arad':-140},

    'Timisoara':{'Lugoj':-111,'Arad':-118},

    'Lugoj':{'Timisoara':-111,'Mehadia':-70},

    'Rimnicu Vilcea':{'Sibiu':-80,'Craiova':-146,'Pitesti':-97},

    'Mehadia':{'Lugoj':-70,'Drobeta':-75},

    'Drobeta':{'Mehadia':-75,'Craiova':-120},

    'Pitesti':{'Rimnicu Vilcea':-97,'Craiova':-138,'Bucharest':-101},

    'Craiova':{'Drobeta':-120,'Rimnicu Vilcea':-146,'Pitesti':-138},

    'Fagaraw':{'Sibiu':-99,'Bucharest':-211},

    'Bucharest':{'Pitesti':-101,'Giurgin':-90,'Fagaraw':-211,'Urziceni':-85},

    'Giurgin':{'Bucharest':-90},

    'Urziceni':{'Bucharest':-85,'Hirsova':-98,'Vaslui':-142},

    'Eforie':{'Hirsova':-86},

    'Hirsova':{'Eforie':-86,'Urziceni':-98},

    'Vaslui':{'Urziceni':-142,'Iasi':-92},

    'Iasi':{'Vaslui':-92,'Neamt':-87},

    'Neamt':{'Iasi':-87}

}

import heapq

def ucs(graph, start, goal):

    frontier = [(0, start)]

    explored = set()

    while frontier:

        current\_cost, current\_node = heapq.heappop(frontier)

        if current\_node == goal:

            return current\_cost

        explored.add(current\_node)

        for neighbor, cost in graph[current\_node].items():

            new\_cost = current\_cost + cost

            if neighbor not in explored:

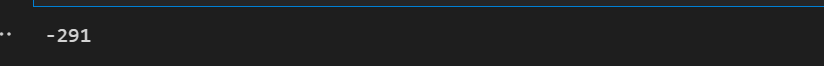
                frontier\_cost = next((c for (c, n) in frontier if n == neighbor), None)

                if frontier\_cost is None or new\_cost < frontier\_cost:

                    heapq.heappush(frontier, (new\_cost, neighbor))

    return None

ucs(graph=graph,start='Oradea',goal='Arad')



graph = {

    'Oradea':{'Zerind':71,'Sibiu':-151},

    'Zerind':{'Arad':75,'Oradea':-71},

    'Arad':{'Timisoara':118,'Sibiu':140,'Zerind':75},

    'Sibiu':{'Fagaraw':99,'Rimnicu Vilcea':80,'Oradea':151,'Arad':140},

    'Timisoara':{'Lugoj':-111,'Arad':118},

    'Lugoj':{'Timisoara':111,'Mehadia':70},

    'Rimnicu Vilcea':{'Sibiu':80,'Craiova':146,'Pitesti':97},

    'Mehadia':{'Lugoj':70,'Drobeta':-75},

    'Drobeta':{'Mehadia':75,'Craiova':-120},

    'Pitesti':{'Rimnicu Vilcea':-97,'Craiova':138,'Bucharest':-101},

    'Craiova':{'Drobeta':120,'Rimnicu Vilcea':146,'Pitesti':138},

    'Fagaraw':{'Sibiu':99,'Bucharest':-211},

    'Bucharest':{'Pitesti':101,'Giurgin':90,'Fagaraw':-211,'Urziceni':85},

    'Giurgin':{'Bucharest':90},

    'Urziceni':{'Bucharest':85,'Hirsova':-98,'Vaslui':-142},

    'Eforie':{'Hirsova':-86},

    'Hirsova':{'Eforie':86,'Urziceni':98},

    'Vaslui':{'Urziceni':-142,'Iasi':92},

    'Iasi':{'Vaslui':92,'Neamt':-87},

    'Neamt':{'Iasi':87}

}

import heapq

def ucs(graph, start, goal):

    frontier = [(0, start)]

    explored = set()

    while frontier:

        current\_cost, current\_node = heapq.heappop(frontier)

        if current\_node == goal:

            return current\_cost

        explored.add(current\_node)

        for neighbor, cost in graph[current\_node].items():

            new\_cost = current\_cost + cost

            if neighbor not in explored:

                frontier\_cost = next((c for (c, n) in frontier if n == neighbor), None)

                if frontier\_cost is None or new\_cost < frontier\_cost:

                    heapq.heappush(frontier, (new\_cost, neighbor))

    return None

ucs(graph=graph,start='Fagaraw',goal='Lugoj')



**Task 6**

A \* algorithm

import math

from queue import PriorityQueue

class Graph:

    def \_\_init\_\_(self):

        self.nodes = {

            'A': (0, 0),

            'B': (0, 2),

            'C': (2, 0),

            'D': (2, 2),

            'E': (4, 0),

            'F': (4, 2),

            'G': (6, 1)

        }

        self.edges = {

            'A': {'B': 2, 'C': 5},

            'B': {'A': 2, 'D': 3},

            'C': {'A': 5, 'D': 2, 'E': 4},

            'D': {'B': 3, 'C': 2, 'F': 1},

            'E': {'C': 4, 'F': 3},

            'F': {'D': 1, 'E': 3, 'G': 2},

            'G': {'F': 2}

        }

    def neighbors(self, node):

        return self.edges[node].keys()

    def cost(self, a, b):

        return self.edges[a][b]

def euclidean\_distance(a, b, nodes):

    ax, ay = nodes[a]

    bx, by = nodes[b]

    return math.sqrt((bx-ax)\*\*2 + (by-ay)\*\*2)

def astar\_search(start, goal, graph):

    frontier = PriorityQueue()

    frontier.put(start, 0)

    came\_from = {}

    cost\_so\_far = {}

    came\_from[start] = None

    cost\_so\_far[start] = 0

    while not frontier.empty():

        current = frontier.get()

        if current == goal:

            break

        for next\_node in graph.neighbors(current):

            new\_cost = cost\_so\_far[current] + graph.cost(current, next\_node)

            if next\_node not in cost\_so\_far or new\_cost < cost\_so\_far[next\_node]:

                cost\_so\_far[next\_node] = new\_cost

                priority = new\_cost + \

                    euclidean\_distance(next\_node, goal, graph.nodes)

                frontier.put(next\_node, priority)

                came\_from[next\_node] = current

    return came\_from, cost\_so\_far

if \_\_name\_\_ == '\_\_main\_\_':

    graph = Graph()

    start = 'A'

    goal = 'G'

    came\_from, cost\_so\_far = astar\_search(start, goal, graph)

    # Print the shortest path found

    path = [goal]

    node = goal

    while node != start:

        node = came\_from[node]

        path.append(node)

    path.reverse()

    print(path)

    print("Cost is " + str(cost\_so\_far[goal]))

