

2016/2017

**Programming, Algorithms and Data Structures (210CT)**

Coursework

**MODULE LEADER:** Dr. Diana Hintea

**Student Name: Hannah Rice**

**SID: 6238519**

1. **can confirm that all work submitted is my own: Yes**

**Question 1:**

import random

import sys

def random\_shuffle(A,shuffled,length):

""" Function to randomly shuffle an array of integers """

a = random.choice(A) # Random value from list A

while length > 0: # Base Case for recursion

if a in shuffled: # Cannot have duplicate values

A.remove(a)

random\_shuffle(A,shuffled,length)

return shuffled

else:

shuffled.append(a) # Add randomly chosen value

length = length-1 # Moves towards base case

random\_shuffle(A,shuffled,length)

return shuffled

try: # Check for invalid input to avoid errors.

A = ([1,3,4,6,7,2])

shuffled = [] # New list to store randomly shuffled input

length = len(A)

random\_shuffle(A,shuffled,length)

except NameError:

print("Invalid input: Must be integers")

sys.exit()

except IndexError:

print("Invalid input: Must only be one of each number")

sys.exit()

print(shuffled)

**Explanation:**

I used a recursive function to randomly choose a value in A and then add that value into new array. I used a while loop to ensure that the new array would be the same length as the input. To ensure that each value in the array would only be used once and to increase the efficiency I decided to remove the value that had been selected from the input once it had been added to the new array. I added except statements to ensure that if invalid input was given there would be no errors. I avoided infinite recursion by with every call decreasing the lengths of the array.

**Question 2:**

import sys

def factorial\_trailing\_zeroes(n):

""" Function first determines the factorial of input """

""" Then calculates the number of trailing zeroes """

if n > 0: # A factorial number must be greater than 0

factorial = 1

for i in range(1, n+1): # Loop range determined by input

factorial = (factorial\*i) # Determines the factorial of input

# (1\*1 = 1), (1\*2 = 2), (2\*3 = 6), (6\*4 = 24), etc...

calc = (5\*\*(i-1)) # Determines trailing zeroes

# eg: (5\*\*0, 5\*\*1, 5\*\*2, 5\*\*3, 5\*\*4)

print(calc)

calc = int(calc)

ans = (factorial/calc)

ans = int(ans)

trailing\_zeroes = (ans + 1)

print(str(n) + "!" + " = " + str(factorial))

print("The number of trailing zeroes are " + str(trailing\_zeroes))

else:

print("Input must be greater than zero")

try: # Catch Input Errors

n = (5)

factorial\_trailing\_zeroes(n)

except NameError:

print("Invalid Input: Must be a whole number")

sys.exit()

except TypeError:

print("Invalid Input: Must be an integer")

sys.exit()

**Question 3:**

**Pseudocode:**

HIGHEST\_PERFECT\_SQUARE (n)

ans <- 0

if n >= 0

while ans\*ans < n

ans <- ans + 1

if ans\*ans = n

return(n)

else

n <- (n-1)

repeat(n)

else:

return n

**Python:**

import sys

def highest\_perfect\_square(enter):

""" Returns the highest perfect square which is less than or equal to input """

factor = 0

if enter >= 0:

while factor\*factor < enter: # Next highest to input

factor = factor + 1 # 1\*1, 2\*2, 3\*3, etc

if factor\*factor == enter: # Either the first input or less than n

print(str(enter) + " is a perfect square number")

else:

enter = enter-1 # Counts down from n, until perfect square

highest\_perfect\_square(enter)

else:

print("invalid input")

try: # Catch for input errors

enter = (36)

highest\_perfect\_square(enter)

except NameError:

print("Invalid Input")

sys.exit()N

except TypeError:

print("Invalid Input")

sys.exit()

**Question 4:**

Random\_shuffle = O(n)

Pseudocode example of algorithm:

RANDOM\_SHUFFLE(A, SHUFFLED, LENGTH)

A -> random.choice(A) 1

WHILE length > 0 n

If a in shuffled n

A.remove(a) n

return shuffled 1

else n

shuffled.append(a) n

length -> length – 1 n

return shuffled 1

Factorial\_Trailing\_Zeroes = O(n)

Pseudocode example of algorithm:

FACTORIAL\_TRAILING\_ZEROES(n)

If n > 0 1

factorial -> 1 1

for i in range(1, n+1) n

factorial -> (factorial\*i) n

calc -> (5\*\*(i-1)) n

ans -> (factorial / calc) n

trailing\_zeroes -> (ans + 1) n

The performance of these algorithms will drop linearly depending on the size of the input. This is because there are no nested loops, which are exponentially expensive and we assume the worst case for input size. The most expensive part of the algorithms are the loops, because we don’t know how many times they will iterate.

**Question 5:**

**Pseudocode:**

MATRIX\_ADDITION(B,C)

for i in range(len(B)) n

for j in range(len(B[i])) n\*n

A[i][j] <- B[i][j] + C[i][j] n\*n

MATRIX\_SUBTRACTION(B,C)

for i in range(len(B)) n

for j in range(len(B[i])) n\*n

A[i][j] <- B[i][j] – C[i][j] n\*n

MATRIX\_MULTIPLICATION(B,C)

for i in range(len(B)) n

for j in range(len(B[i])) n\*n

A[i][j] <- B[i][j] \* C[i][j] n\*n

**Python:**

import sys

# Empty matrix to store answer

answer = [[0,0,0,0], [0,0,0,0], [0,0,0,0], [0,0,0,0]]

# Matricies to store resuts of calculations

calc\_first\_half = [[0,0,0,0], [0,0,0,0], [0,0,0,0], [0,0,0,0]]

calc\_second\_half = [[0,0,0,0], [0,0,0,0], [0,0,0,0], [0,0,0,0]]

def matrix\_addition(calc\_first\_half, B, C):

""" Add all values in matrix B and C, store in calc\_first\_half """

for i in range(len(B)): # Adds each element in B with C

for j in range(len(B[0])):

calc\_first\_half[i][j] = B[i][j] + C[i][j] # Stores result

print("")

print("B + C = ")

print(calc\_first\_half)

matrix\_multiplication\_second(calc\_first\_half,B,C)

def matrix\_subtraction(answer, calc\_first\_half, calc\_second\_half):

""" Subtracts all values in calc\_first\_half with calc\_second\_half """

for i in range(len(B)):

for j in range(len(B[0])):

answer[i][j] = calc\_first\_half[i][j] - calc\_second\_half[i][j]

# Stores result in "answer" which is the result of the given calculation

def matrix\_multiplication(calc\_second\_half, B, C):

""" Multiplies all B values with C values """

for i in range(len(B)):

for j in range(len(B[0])):

calc\_second\_half[i][j] = B[i][j] \* C[i][j]

print("")

print("B \* C = ")

print(calc\_second\_half)

def matrix\_multiplication\_second(calc\_first\_half, B, C):

""" Multiplies B + C by 2 """

for i in range(len(B)):

for j in range(len(B[0])):

calc\_first\_half[i][j] = calc\_first\_half[i][j] \* 2

print("")

print("2 \* (B+C) = ")

print(calc\_first\_half)

try: # Catch Input Errors

# Matricies for calculations

B = [[1,2,3,4], [5,6,7,8], [9,10,1,2]],[3,4,5,6]]

C = [[5,8,2,4], [6,1,8,2], [2,3,4,9], [10,7,9,5]]

if len(B) == len(C): # Length of matrices must be the same

print("A = B\*C - 2\*(B+C) ")

matrix\_addition(calc\_first\_half, B, C)

matrix\_multiplication(calc\_first\_half, B, C)

matrix\_subtraction(answer, calc\_first\_half, calc\_second\_half)

print("")

print("Answer :")

print(answer)

else:

print("The matricies must be the same size")

except NameError:

print("Invalid Value in Matrix")

sys.exit()

**Runtime:**

**Question 6:**

**Pseudocode:**

REVERSE\_SENTENCE(S)

X <- Length(S)

If X <= 0

return S

else

X <- X-1

S <- S

reverse\_sentence(S[1:]) to S[0:]

return S

S <- (“This”, “is”, “awesome”)

reverse\_sentence(S)

**Python:**

import sys

def reverse\_sentence(s):

""" Takes 's' as input then reverses the string """

sLength = len(s)

if sLength <= 0: # Stops infinite recursion

# The function stops once the sentence in reversed

return s

else:

sLength = sLength-1

s = s

reverse\_sentence(s[1:]) + s[0:] # Recalls the function with new '0'

position = s[0] # Saves position 0 value

# S[0] changes each iteration

print(position)

return s # Continues through the input

sentence = ("This is awesome") # Original input

if type(sentence) != str: # Catch for input errors

print("Input must be a string")

sys.exit()

s = sentence.split() # Split so that can index each word in sentence

print(sentence)

print("")

print("REVERSED: ")

print("")

reverse\_sentence(s)

**Question 7:**

**Pseudocode:**

PRIME(n,i)

If n > 1 and i > 1

ans <- n % i

if ans != 0

i <- i – 1

prime(n, i)

return i

elif ans = 0

not a prime number

else

prime number

n <- 3

i <- n – 1

prime(n,i)

**Python:**

import sys

def prime(enter, x):

""" Calculates if the input is a prime number or not """

if enter > 1 and x > 1: # Prime number must be greater than 1

result = enter % x # Enter / 1 but disregards the remainder

if result != 0:

x = x-1 # Checks n / every number below n until reach 0

prime(enter,x)

return x

elif result == 0:

print(str(enter) + " is not a prime number")

else:

print(str(enter) + " is a prime number")

try: # Catch for input errors

enter = 3

if enter > 1:

x = enter - 1

prime(enter,x)

else:

print("Input must be greater than 1")

except NameError:

print("Input must be an integer")

sys.exit()

except TypeError:

print("Input must be an integer")

sys.exit()

**Question 8:**

**Pseudocode:**

REMOVE\_VOWELS(S,V,x,new)

If x > 0

x <- (x-1)

if V[x] in S

a <- V[x]

new <- new(remove(a))

remove\_vowels(S,V,x,new)

return x

else

remove\_vowels(S,V,x,new)

else

return x

S <- (“input”)

V <- (“a”,”e”,”i”,”o”,”u”)

x <- Length(V)

new <- S

output(S)

remove\_vowels(S,V,x,new)

**Python:**

import sys

def remove\_vowels(word, vowels, position, word\_copy):

""" Takes input and returns the input without the vowels """

if position > 0: # Base case

# Itterates through vowels, also moves towards base case

position = position-1

if vowels[position] in word:

# Each call checks if that vowel is in the input

a = vowels[position]

word\_copy = word\_copy.replace(a,"") # Takes out the vowel

print(word\_copy)

remove\_vowels(word,vowels,position,word\_copy)

return position

else: # Moves to next vowel

remove\_vowels(word,vowels,position,word\_copy)

return position

else:

return position

word = ("beautiful")

if type(word) != str: # Check for input errors

print("Input must be a string")

sys.exit()

vowels = ("a","e","i","o","u","A","E","I","O","U")

position = len(vowels)

word\_copy = word

print(word) # If there are no vowels the input word will show

remove\_vowels(word,vowels,position,word\_copy)

**Question 9:**

def binary\_search(entry): # Divide and Conquer

""" Search through input for values within the given high & low parameters """

length = len(entry)

middle = length/2 # Find the middle value in the list

middle = int(middle)

if entry[middle] == low or entry[middle] == high:

print("TRUE")

elif middle < 1: # Value not in range

print("FALSE")

else:

if entry[middle] > low and entry[middle] < high:

print("TRUE")

elif low < entry[middle]:

entry = entry[:middle] # Disregard first half of list

binary\_search(entry)

return entry

elif high > entry[middle]:

entry = entry[middle:] # Disregard second half of list

binary\_search(entry)

return entry

else:

print("FALSE")

try: # Check for input errors

entry = [2,3,5,7,9,13]

low = 10

high = 14

if low < high:

binary\_search(entry)

else:

print("Low parameter must be less than High parameter")

except NameError:

print("Input must be all integers")

**Big O complexity**

**Question 10:**

import sys

class Extract\_Sub\_Sequence:

""" Extract the sub sequence of maximum length in ascending order """

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

self.sequence = sequence

Extract\_Sub\_Sequence.search\_sequence(self)

def search\_sequence(self):

""" Search the sequence values and determine length """

current = (self.sequence[0])

start = sequence[0] # Start search at first element

A = [start]

B = [] # Will store new sub sequences for length comparison

for i in range(len(sequence)):

if self.sequence[i] > self.sequence[i-1]: # Ascending order

current = self.sequence[i]

A.append(current) # Store sub sequence value

print(A)

else:

if len(A) > len(B): # Puts current value into new A[]

B = A

A = [sequence[i]]

if len(A) > len(B): # Determines which sub sequence is longer

print("")

print(A)

else:

print("")

print(B)

try: # Catch input errors

sequence = [1,2,3,4,1,5,1,6,7]

Extract\_Sub\_Sequence(sequence)

except NameError:

print("Sequence must be numbers")

except TypeError:

print("Sequence must be integers not string")

**Question 11:**

class Node:

""" Establishes Node with connected Nodes """

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

self.value = value

self.next = None

self.prev = None

class Double\_Linked\_List:

""" Insert given nodes and remove given nodes to and from a double linked list """

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

self.head = None

self.tail = None

def insert\_node(self, node, point):

""" Insert given node into the linked list """

if node != None:

point.next = node.next # Point next node

node.next = point # New next node is point

point.prev = node # Point to previous node

if point.next != None:

point.next.prev = point

if self.head == None: # When list is empty

self.head = self.tail = point # Head and tail will equal the node

# Nothing to point to because list was empty

point.prev = point.next = None

elif self.tail == node:

self.tail = point

# ------------------------------------------------------------------- |

def remove\_node(self, node):

""" Remove given node from the linked list """

if node.prev != None: # Previous node will point to next node

node.prev.next = node.next

else:

self.head = node.next

if node.next != None: # Next node now equals the previous

node.next.prev = node.prev

else:

self.tail = node.prev

# ------------------------------------------------------------------- |

def show\_list(self):

""" Displays the linked list """

values = [] # Creates the list type

node = self.head

while node != None:

values.append(str(node.value)) # Adds nodes to list

node = node.next

# .join = returns string, elements joined by str operation

print(" ---> ".join(values))

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

l = Double\_Linked\_List()

l.insert\_node(None, Node(4))

l.insert\_node(l.head, Node(6))

l.insert\_node(l.head, Node(8))

l.insert\_node(l.head, Node(3))

l.insert\_node(l.head, Node(2))

l.show\_list()

l.remove\_node(l.head)

l.show\_list()

**Question 12:**

import sys

class Tree:

""" Assigns values and left and right variables for tree structure """

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

self.value = value

self.right = None

self.left = None

def insert\_value(tree, item): # Binary search tree

""" Insert values into Binary Tree structure """

if tree == None:

tree = Tree(item) # Makes values NONE

else:

if (item < tree.value): # Left side must be smaller

if (tree.left == None):

tree.left = Tree(item) # Becomes the new root

else:

insert\_value(tree.left,item)

else: # Right side must be larger

if(tree.right == None):

tree.right = Tree(item)

else:

insert\_value(tree.right, item)

return tree

def in\_order\_iterative(tree\_root):

""" Orders Nodes in Left, Node, Right (in order) structure """

current = tree\_root # Root of tree(top value in tree)

stack = [] # LIFO (last in first out)

while True:

while current != None: # A leaf has no children

stack.append(current) # Add values into stack

current = current.left # Continue moving left

if current == None: # No more children

length = len(stack)

# Stack will only be empty once all nodes have been added to order

if length > 0:

current = stack.pop() # Takes last in stack value

# Value gets removed from stack at the same time

print(current.value)

current = current.right

# Now we have gone left, need to go right

else:

sys.exit() # Stop infinite loop

# Example Graph for Visual representation

t = insert\_value(None,10) # 10

tree\_root = insert\_value(None, None) # / \

tree\_root.left = insert\_value(t, 8) # 8 14

insert\_value(t, 14) # / \ / \

insert\_value(t, 5) # 5 9 12 17

insert\_value(t, 9) # / \

insert\_value(t, 12) # 11 13

insert\_value(t, 17)

insert\_value(t, 11)

insert\_value(t, 13)

in\_order\_iterative(tree\_root)

**Question 13:**

**Pseudocode:**

GRAPH

vertices -> {}

ADD\_VERTEX(v)

vertices[v] -> []

ADD\_EDGE(e)

vertices(v, []) (e)

return vertices

**Python:**

# Graph = {'1' : ['2'], # Adjacency List

# '2' : ['1','4'],

# '3' : ['4','5'],

# '4' : ['2','3','5'],

# '5' : ['3','4'] }

class Graph:

""" Insert given vertex and edge into graph """

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

vertices = {} # Used a Dictionary to store vertex as key

# and lists as values

self.vertices = vertices

def add\_vertex(self,new\_vertex): # Add vertex = add key

""" Add vertex as key """

self.vertices[new\_vertex] = []

def add\_edge(self, new\_vertex, new\_edge):

""" Add edge values in list """

self.vertices.setdefault(new\_vertex, []).append(new\_edge)

def show\_graph(self):

""" Display the graph """

print("")

print(self.vertices)

g = Graph()

g.add\_vertex('1')

g.add\_vertex('2')

g.add\_vertex('3')

g.add\_vertex('4')

g.add\_vertex('5')

g.show\_graph()

g.add\_edge('1',['2','6'])

g.add\_edge('2',['1','4'])

g.add\_edge('3',['4','5'])

g.add\_edge('4',['2','3','5','6'])

g.add\_edge('5',['3','4','6'])

print("")

print("Graph")

g.show\_graph()

**Question 14:**

**Depth First Search:**

import sys

class Graph:

""" Insert given vertex and edge into graph """

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

vertices = {} # Used a Dictionary to store vertex as key

# - and lists as values

visited = [] # Empty list for DFS

self.vertices = vertices

self.visited = visited

def add\_vertex(self, new\_vertex):

""" Add vertex as key """

self.vertices[new\_vertex] = []

def add\_edge(self, new\_vertex, new\_edge):

""" Add edge to values in list """

self.vertices.setdefault(new\_vertex,[]).append(new\_edge)

def show\_graph(self):

""" Display the graph """

print(self.vertices)

def depth\_first\_search(self, start\_search, stack, z):

""" Depth first search of graph """

try: # Catch index error

if start\_search not in self.visited: # Not visit the same vertex

self.visited.append(start\_search)

# Add current vertex to visited

print("Visited")

print(self.visited)

search = self.vertices[start\_search]

# Access key(vertex),values(edges)

x = 0

go = True

while go == True: # Itterate over edges and add to stack

for a,v in enumerate(search):

# Allows to index values eg:(0,'2')

try: # Catch list index error

if v[x] not in stack: # No duplicates in stack

stack.append(v[x])

x = x+1

if v[x] in self.visited:

# Remove value from visited

stack.remove(v[x])

else:

x = x+1

except:

go = False # Break loop

print("Stack")

print(stack)

print("")

start\_search = stack.pop(z) # Take last in value(edge)

# Becomes current vertex

Graph.depth\_first\_search(self, start\_search, stack, z)

return start\_search

elif start\_search in self.visited:

# If we have already visited the vertex

start\_search = stack.pop(z)

# Take next edge in stack

stack.clear() # Empty stack to not skip vertexes

Graph.depth\_first\_search(self, start\_search, stack, z)

else:

return start\_search

except:

print("Depth First Search : ")

print(self.visited)

g = Graph()

g.add\_vertex('A') # Add vertexes

g.add\_vertex('B')

g.add\_vertex('C')

g.add\_vertex('D')

g.add\_vertex('E')

g.add\_vertex('F')

g.add\_vertex('G')

g.add\_vertex('H')

g.add\_vertex('S')

g.show\_graph()

print("")

g.add\_edge('A',['B','S']) # Add Edges

g.add\_edge('B',['A'])

g.add\_edge('S',['A','C','G'])

g.add\_edge('C',['S','F','E','D'])

g.add\_edge('D',['C'])

g.add\_edge('E',['C','H'])

g.add\_edge('F',['G','C'])

g.add\_edge('G',['S','H','F'])

g.add\_edge('H',['G','E'])

g.show\_graph()

start\_search = ('A') # Start node for traversal

z = 0 # Allows to access position 0 of stack for pop()

stack = [] # Empty stack

g.depth\_first\_search(start\_search, stack, z)

**Breadth First Search:**

import sys

class Graph:

""" Insert given vertex and edge into graph """

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

vertices = {} # Used a Dictionary to store vertex as key

# - and lists as values

self.vertices = vertices

visited = [] # Empty list for BFS

self.visited = visited

def add\_vertex(self, new\_vertex):

""" Add vertex as key """

self.vertices[new\_vertex] = []

def add\_edge(self, new\_vertex, new\_edge):

""" Add edge to values in list """

self.vertices.setdefault(new\_vertex,[]).append(new\_edge)

def show\_graph(self):

""" Display the graph """

print(self.vertices)

def breadth\_first\_search(self, start\_search, z, queue):

""" Breadth first search of graph """

try: # Catch index error

if start\_search not in self.visited:

# Don't want to re-visit vertex

self.visited.append(start\_search)

# Add current vertex to 'Visited'

print("Visited")

print(self.visited)

search = self.vertices[start\_search]

# Get values linked to current vertex

x = 0

go = True

while go == True:

for a,v in enumerate(search):

# Access each list element separately

try:

if v[x] not in queue:

# Don't want duplicate edges in queue

queue.append(v[x]) # Add edge to queue

x = x +1

else:

x = x+1

except IndexError:

print("Queue")

print(queue)

print("")

go = False # Breaks loop

start\_search = queue[z]

# Next current vertex is first item in queue

Graph.breadth\_first\_search(self, start\_search, z, queue)

elif start\_search in self.visited:

# If already been to that vertex

queue.remove(start\_search)

# Remove vertex from queue, because we have already been there

start\_search = queue[z]

Graph.breadth\_first\_search(self,start\_search,z, queue)

else:

return start\_search

except IndexError:

print("Breadth first search : ")

print(self.visited)

g = Graph() # Add vertex

g.add\_vertex('A')

g.add\_vertex('B')

g.add\_vertex('C')

g.add\_vertex('D')

g.add\_vertex('E')

g.add\_vertex('F')

g.add\_vertex('G')

g.add\_vertex('H')

g.add\_vertex('S')

g.show\_graph()

print("")

g.add\_edge('A',['B','S']) # Add edges

g.add\_edge('B',['A'])

g.add\_edge('S',['A','C','G'])

g.add\_edge('C',['D','E','F','S'])

g.add\_edge('D',['C'])

g.add\_edge('E',['C','H'])

g.add\_edge('F',['C','G'])

g.add\_edge('G',['S','F','H'])

g.add\_edge('H',['G','E'])

g.show\_graph()

z = 0 # Used to access position 0 in the queue

queue = [] # Start with empty queue

start\_search = ('A') # Start search from vertex 'A'

g.breadth\_first\_search(start\_search, z, queue)

**Question 15:**

#my\_graph = {'1' : ['2'(5)],

# '2' : ['1'(5),'4'(7)],

# '3' : ['4'(4),'5'(8)],

# '4' : ['2'(7),'3'(4),'5'(3)],

# '5' : ['3'(8),'4'(3)] }

import sys

class Graph:

""" Insert given vertex and edges into graph """

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

vertices = {} # Used a dictionary to store vertex as key

self.vertices = vertices

visited = []

self.visited = visited

def add\_vertex(self,new\_vertex):

""" Add vertex as key """

self.vertices[new\_vertex] = []

def add\_edge(self, new\_vertex, new\_edge, weight):

""" Add edge and weight """

self.vertices.setdefault(new\_vertex, []).append(new\_edge)

self.vertices[new\_vertex].append(weight)

def show\_graph(self):

""" Display the graph """

print(self.vertices)

def dijkstra(self, start, destination, visited, weight):

""" Find the shortest path using the weights of each edge """

edge = []

if start not in visited: # If not already visited vertex

if start != destination:

visited.append(start) # Add current vertex to visited list

print("")

print("Visited")

print(visited)

search = self.vertices[start]# Access edges to current vertex

for w in search: # Find the weights

if type(w) == int: # Weights are integers

weight.append(w) # Put weights into list

else:

edge.append(w) # Put edges into list

print("")

min\_weight = min(weight) # Find lowest weight from list

weight.clear()

weight.append(min\_weight) # Only store the min weight

# The edge before min weight will be our next start vertex

if weight[0] == search[1]:

a = str(edge[0])

start = a

g.dijkstra(start, destination, visited, weight)

return start

elif weight[0] == search[3]:

start = edge[1]

g.dijkstra(start, destination, visited, weight)

return start

elif weight[0] == search[5]:

start = edge[2]

g.dijkstra(start, destination, visited, weight)

return start

else:

return start

else:

print("The shortest path : ")

print(visited)

sys.exit()

g = Graph() # Graph for visual representation:

g.add\_vertex('1') # 1 --- 2 --- 4 --- 5

g.add\_vertex('2') # | /

g.add\_vertex('3') # 3 - /

g.add\_vertex('4')

g.add\_vertex('5')

g.show\_graph()

print("")

weight = (5) # Add edges and weights

g.add\_edge('1',('2'),weight)

weight = (5)

g.add\_edge('2',('1'), weight)

weight = (1)

g.add\_edge('2',('4'), weight)

weight = (3)

g.add\_edge('3',('4'), weight)

weight = (2)

g.add\_edge('3',('5'), weight)

weight = (1)

g.add\_edge('4',('2'), weight)

weight = (3)

g.add\_edge('4',('3'), weight)

weight = (6)

g.add\_edge('4',('5'), weight)

weight = (2)

g.add\_edge('5',('3'), weight)

weight = (6)

g.add\_edge('5',('4'), weight)

g.show\_graph()

start = '1' # Start path

destination = '3' # End path

weight = []

visited = []

g.dijkstra(start, destination, visited, weight)