**Week 1**

(Task 1 Basic)

**from** random **import** randint  
  
list\_1 **= [**5,3,8,6,1,9,2,7**]** # The list that has to be randomized  
list\_2 **= [**i **for** i **in** range**(**len**(**list\_1**))]** # Creates a second list with the same size  
count**=**0  
filled\_index**=[]** # Stores the indexes already taken in the second list  
  
**while** count**!=**len**(**list\_1**):** # Stops when all of the elements of the first list have been added to the second  
 el **=** randint**(**0,**(**len**(**list\_1**) -** 1**))** # Generates a random index from the second list  
 **if** el **not in** filled\_index**:** # Checks if that index has already been taken in order to insert   
 filled\_index.append**(**el**)** # Updates the taken indexes  
 list\_2**[**el**] =** list\_1**[**count**]** # Inserts from the first list into the free index of the second list  
 count **=** count **+** 1

print **("Original list : "**,list\_1**)**print **("Shuffled list : "**,list\_2**)**

(Task 2 Basic)

number **=** int**(**input**("Enter a number: "))  
  
for** i **in** range**(**number, 0, **-**1**):** # Calculates the factorial number  
 number **=** number **\*** i  
  
print **("The factorial number is : "**, number**)**number **=** list**(** map**(**int, str**(**number**)))** # Makes a list of the digits in the number  
  
count **=** len**(**number**)  
while True:** # Counts the number of trailing 0s in the end of the factorial number  
 **if** number**[**count **-** 1**] ==** 0**:** # Iterates over the digits in the factorial number starting from the end  
 count **-=** 1  
 **else:** print**("The number of trailing 0s is the factorial number is : "**, int**(**len**(**number**)) -** count**)  
 break**

**Week 2**

(Task 3 Basic)

Pseudocode:

PERFECT\_SQUARE(number)

digit 🡨 0

perfect\_square 🡨 none

while TRUE // Incrementing a digit starting from 1 until it's perfect square

digit 🡨 digit + digit // is more than the given number and prints the previous perfect square

if digit\*\*2 < number

perfect\_square 🡨 digit\*\*2

else

return perfect\_square

Code:

number **=** int**(**input**("Enter a number to find the closest perfect square which is less or equal to it : "))**digit **=** 0  
perfect\_square **= None  
  
while True:** # Incrementing a digit starting from 1 until it's perfect square   
 digit **+=** 1 # is more than the given number and prints the previous perfect square  
 **if** digit**\*\***2 **<** number**:** perfect\_square **=** digit**\*\***2  
 **else:** print **("The closest perfect square is : "**, perfect\_square**)  
 break**

(Task 4 Basic)

#Week\_1 Task\_1 Big(O) notation

from random import randint O(1)

list\_1 = [5,3,8,6,1,9,2,7] O(1)

count=0 O(1)

filled\_index=[] O(1)

while count!=len(list\_1): O(n)

el = randint(0,(len(list\_1) - 1)) O(n)

if el not in filled\_index: O(n?)

filled\_index.append(el) O(n?)

list\_2[el] = list\_1[count] O(n?)

count = count + 1 O(n?)

print ("Original list : ",list\_1) O(1)

print ("Shuffled list : ",list\_2) O(1)

The Big(o) is : O(n)

#Week\_1 Task\_2 Big(O) Notation

number = int(input("Enter a number: ")) O(1)

for i in range(number, 0, -1): O(n)

number = number \* I O(n)

print ("The factorial number is : ", number) O(1)

number = list( map(int, str(number))) O(1)

count = len(number) O(1)

while True: O(n)

if number[count - 1] == 0: O(n?)

count -= 1 O(n?)

else: O(n?)

print("The number of trailing 0s is the factorial number is : ", int(len(number)) - count) O(n?)

break

O(n?)

The Big(o) is : O(n)

(Task 5 Basic)

**ADDITION\_SUBTRACTION (A,B,C,r\_c,p\_m) # A and B matrix to multiply,C empty matrix , (r\_c) rol and coll of the matrix, (p\_m)plus or minus  
  
 for i<--0 to r\_c.size()  
 for j<--o to r\_c.size()  
 if p\_m = "+"  
 C[i][j] = A[i][j] + B[i][j]  
 elseif p\_m = "-'  
 C[i][j] = A[i][j] - B[i][j]  
 return C  
   
MULTIPLY\_M (A,B,C,r\_c) // Multiply 2 matrix  
 count\_1 <-- count\_2 <-- 0  
 for i<--1 to r\_c.size() + 1  
 count\_1 ++  
 for j<--1 to r\_c.size() + 1  
 count\_2 ++  
 for p<--0 to r\_c.size()  
 C[i][j] <-- A[j - count\_2][i + p - count\_1] \* B [i + p][j - count\_1] + C[i][j]  
 return C  
  
  
  
// compute A=B\*C –2\*(B+C)  
   
A <-- ADDITION (MULTIPLY\_M(B,C,D,r\_c), (2 \* ADDITION(A,B,Q,r\_c,"+"), P, r\_c, "-") // D,P and Q new matrix that stores result**

**Week 3**

(Task 6 Basic)

**"""  
REVERSE\_SENTENCE(sentence)  
 sentence <-- sentence + " " O(1)  
 container <-- () -  
 start <-- 0 -  
 end <-- 0 -  
 rev\_sent <-- '' O(1)  
  
 while end < length[sentence] O(n)  
 if sentence[end] <-- " " O(n?)  
 container.add(sentence[start:end + 1]) -  
 end <-- end + 1 -  
 start <-- end O(n?)  
 end <-- end + 1 O(n)  
  
 for i <-- length(container)-1 to -1 by -1 O(n)  
 rev\_sent <-- rev\_sent + container[i] O(n)  
 return rev\_sent O(1)  
  
 Big (O) Notation : O(n)  
  
  
"""  
  
  
def reverse\_sentace(**sentence**):** sentence **=** sentence **+ " "** #adds a white space in the end of the sentance  
 container **= []** start **=** 0  
 end **=** 0  
 rev\_sent **= ""  
 while** end **<** len**(**sentence**):** # Makes the string input a list of strings  
 **if** sentence**[**end**] == " ":** container.append**(**sentence**[**start**:**end **+** 1**])** end **+=** 1  
 start **=** end  
 end **+=** 1  
  
 **for** i **in** range**((**len**(**container**)-**1**)**,**-**1, **-**1**):** # Turns the list of words around  
 rev\_sent **=** rev\_sent **+** container**[**i**]** print **(**rev\_sent**)**reverse\_sentace**("This is awesome")**

(Task 7 Basic)

**"""  
PRIME\_NUMBER\_CHECK(number, count <-- 2)  
 check <-- number % count  
 if check = 0 and number != count or number = 1  
 OUTPUT "It is not a prime number!"  
 return  
 elseif number = check  
 OUTPUT "It is a prime number!"  
 return  
  
 count <-- count + 1  
 PRIME\_NUMBER\_CHECK(number, count)  
"""  
def prime\_number\_check(**number,count **=** 2**):** check **=** number **%** count  
 **if** check **==** 0 **and** number **!=** count **or** number **==** 1**:** print **( "It's not a prime number")  
 return  
 elif** number **==** check**:** print **("It is a prime number")  
 return** print **(**count**)** count**+=**1  
 prime\_number\_check**(**number,count**)**prime\_number\_check**(**199**)**

(Task 8 Basic)

**"""  
REMOVE\_VOWELS(word <-- None, i <-- 0, vowels <-- ('a','e', 'i','o','u') // (i) the index of the current letter  
 if word [i].lowercase() in vowels // Checks if the letter is a vowel and if t is it removes it  
 remove w[i]  
 i <-- i - 1  
 if i = length[word] - 1  
 return word  
 REMOVE\_VOWELS(word, i +1)  
  
"""  
  
def remove\_vowels(**word, i**=**0,vowels **= ('a'**,**'e'**, **'i'**,**'o'**,**'u')):** # (i) the index of the current letter  
 **if** word**[**i**]**.lower**() in** vowels**:** # Checks if the letter is a vowel and if t is it removes it  
 word **=** word.replace**(**word**[**i**]**,**"")** i **-=** 1  
 **if** i **==** len**(**word**) -**1**:** print **(**word**)  
 return** remove\_vowels**(**word,i**+**1**)**remove\_vowels**("beautiful")**

Week 4

(Task 9 Basic)

**"""  
BINARY\_SEARCH\_RANGE(alist, range\_f, range\_l) // Searches for a value within a specific interval (range\_f = range from -/- range\_l = range last  
 count <-- 0  
 lst <-- alist.copy() // Makes a copy of the list as it gets altered  
 lst.add(None)  
 first <-- 0  
 last <-- len(lst) - 1  
 fount <-- TRUE  
  
 while first <= last and fount != TRUE  
 midpoint <-- (first + last) //2 // Gets the middle of the list  
 if last[midpoint] in range\_f to (range\_l+1) // If the middle is in the range return True  
 found <-- TRUE  
 else:  
 if lst[midpoint] < range\_f // Depending on the middle point ( smaller or larger than the range) splits the list in two  
 first <-- midpoint  
 else  
 last <-- midpoint  
  
 count <-- count + 1  
 if count > length[alist] / 2  
 return found  
  
"""  
  
def binarySearch\_range(**alist, range\_f, range\_l**):** # Searches for a value within a specific interval (range\_f = range from -/- range\_l = range last  
 count **=** 0  
 lst **=** alist.copy**()** # Makes a copy of the list as it gets altered  
 lst.append**(None)** first **=** 0  
 last **=** len**(**lst**)-**1  
 found **= False  
  
 while** first**<=**last **and not** found**:** midpoint **= (**first **+** last**) //**2 # Gets the middle of the list  
 **if** lst**[**midpoint**] in** range**(**range\_f, **(**range\_l**+**1**)):** # If the middle is in the range return True  
 found **= True  
 else:  
 if** lst**[**midpoint**] <** range\_f**:** # Depending on the middle point ( smaller or larger than the range) splits the list in two  
 first **=** midpoint  
 **else:** last **=** midpoint  
 count **=** count **+**1  
 **if** count **> (**len**(**alist**)/** 2**):  
 break  
  
  
 return** found  
# 0 1 2 3 4 5 6 7 8 9 10  
lstt **= [**1,3,4,9,14,18,27,35,62,75,81**]**print **(**lstt**)**range\_f **=**int**(**input**("Range from : "))**range\_l **=** int**(**input**("Range to : "))**print **(**binarySearch\_range**(**lstt,range\_f,range\_l**))**

Week 5

(Task 10 Basic)

**def biggest\_sequence (**lst,count**=**0,output**=[]**, A**=**0**):  
  
 if** A **>= (**len**(**lst**) -** 1**):** print**(**max**(**output,key**=**len**))** # max applies the key function to each element of your list and picks the one where the key function returned the biggest  
 **return** output.append**([])  
  
 while True:** i **=** lst**[**A**]  
 if** i **!=** lst**[**len**(**lst**)-**1**] and** i **<** lst**[**lst.index**(**i**)+**1**]:** output**[**count**]**.append**(**i**)** A **+=**1  
 **else:** output**[**count**]**.append**(**i**)** A **+=**1  
 count **+=** 1  
 biggest\_sequence**(**lst,count,output,A**)  
 return**sequence **= [**3,4,6,12,1,7,3,25,28,29,50,30**]**biggest\_sequence**(**sequence**)**

(Task 11 Basic)

**class Node(**object**):  
 def \_\_init\_\_(**self, value**):** self.value **=** value  
 self.next **= None** self.prev **= None  
  
  
class List(**object**):  
 def \_\_init\_\_(**self**):** self.head **= None** self.tail **= None  
  
 def insert(**self, n, x**):** #n is where too insert it (head or tail) // x is the value of the node we are inserting  
 # Not actually perfect: how do we prepend to an existing list?  
 **if** n **!= None:** x.next **=** n.next #1  
 n.next **=** x #2  
 x.prev **=** n #  
 **if** x.next **!= None:** x.next.prev **=** x  
 **if** self.head **== None:** self.head **=** self.tail **=** x  
 x.prev **=** x.next **= None  
 elif** self.tail **==** n**:** self.tail **=** x  
  
 **def display(**self**):** values **= []** n **=** self.head  
 **while** n **!= None:** values.append**(**str**(**n.value**))** n **=** n.next  
 print **("List: "**, **","**.join**(**values**))  
  
 def remove\_value(**self, node\_value**):** current\_item **=** self.head  
 **while** current\_item **is not None:  
 if** current\_item.value **==** node\_value**:  
 if** current\_item.prev **is not None:** current\_item.prev.next **=** current\_item.next  
 current\_item.next.prev **=** current\_item.prev  
 **else:** self.head **=** current\_item.next  
 current\_item.next.prev **= None** current\_item **=** current\_item.next  
  
**if** \_\_name\_\_ **== '\_\_main\_\_':** l **=** List**()** l.insert**(None**, Node**(**4**))** l.insert**(**l.head, Node**(**6**))** l.insert**(**l.head, Node**(**8**))** l.insert**(**l.tail, Node**(**9**))** l.remove\_value**(**4**)** l.display**()**

Week 6

(Task 12 Basic)

**class BinTreeNode(**object**):  
 def \_\_init\_\_(**self, value**):** self.value **=** value  
 self.left **= None** self.right **= None  
  
  
def tree\_insert(**tree, item**):  
 if** tree **== None:** tree **=** BinTreeNode**(**item**)  
 else:  
 if (**item **<** tree.value**):  
 if (**tree.left **== None):** tree.left **=** BinTreeNode**(**item**)  
 else:** tree\_insert**(**tree.left, item**)  
 else:  
 if (**tree.right **== None):** tree.right **=** BinTreeNode**(**item**)  
 else:** tree\_insert**(**tree.right, item**)  
 return** tree  
  
  
**def postorder(**tree**):  
 if (**tree.left **!= None):** postorder**(**tree.left**)  
 if (**tree.right **!= None):** postorder**(**tree.right**)** print **(**tree.value**)  
  
  
def in\_order\_recursive(**tree**):  
 if (**tree.left **!= None):** in\_order\_recursive**(**tree.left**)** print **(**tree.value**)  
 if (**tree.right **!= None):** in\_order\_recursive**(**tree.right**)  
  
def in\_order\_iterative(**tree**):** finished **= False** stack **= []** current\_node **=** tree  
  
 **while (**finished **!= True):  
  
 if** current\_node **!= None:** stack.append**(**current\_node**)** current\_node**=** current\_node.left  
 **else:  
 try:** #Using try because if the list is empty the code will break  
 current\_node **=** stack**[**len**(**stack**) -** 1**]** #Get pointer back to the previous node  
 print**(**current\_node.value**)** # Print the last node of the branch  
 current\_node **=** current\_node.right #Continue on its right  
 stack.pop**()** #Removes the last added node  
 **except:** finished **= True  
  
  
  
  
  
if** \_\_name\_\_ **== '\_\_main\_\_':** t **=** tree\_insert**(None**, 6**)** tree\_insert**(**t, 10**)** tree\_insert**(**t, 5**)** tree\_insert**(**t, 2**)** tree\_insert**(**t, 3**)** tree\_insert**(**t, 4**)** tree\_insert**(**t, 11**)** print**("Recursively print of the tree:")** in\_order\_recursive**(**t**)** print **("Iteratively print of the tree:")** in\_order\_iterative**(**t**)**

Week 7

(Task 13 Basic)

**"""  
CLASS NODE  
 value <-- val  
   
CLASS GRAPH  
 dict <-- new dictionary  
 key <-- []  
   
 ADD\_VERTEX  
 NODE(val)  
 dict val <-- []  
 keys.append(val)  
   
 ADD\_EDGE(vertex\_1, vertex\_2)  
 dict vertex\_1.append(vertex\_2)  
 dict vertex\_2.append(vertex\_1)  
   
"""  
class Node:  
 def \_\_init\_\_(**self,value**):** self.value **=** value  
  
**class Graph:** dict **= {}** # key(nodeNameVale): value( Edges) //edges added here are for the ease of the use on BFS and DFS  
 keys **= []** # Stores the keys for the dictionary  
  
 **def add\_vertex(**self,value**):** # Creates the new Vertex  
 Node**(**value**)** self.dict**[**value**] = []** self.keys.append**(**value**)  
  
 def add\_edge(**self,vertex\_1,vertex\_2**):** # Adds the edge connecting vertex\_1 and vertex\_2  
 self.dict**[**vertex\_1**]**.append**(**vertex\_2**)** self.dict**[**vertex\_2**]**.append**(**vertex\_1**)  
  
  
if** \_\_name\_\_ **=='\_\_main\_\_':** G **=** Graph**()** G.add\_vertex**(**1**)** G.add\_vertex**(**2**)** G.add\_vertex**(**3**)** G.add\_vertex**(**4**)** G.add\_vertex**(**5**)** G.add\_vertex**(**6**)** G.add\_vertex**(**7**)** G.add\_vertex**(**8**)** G.add\_vertex**(**9**)** G.add\_vertex**(**10**)** G.add\_edge**(**1, 2**)** G.add\_edge**(**1, 3**)** G.add\_edge**(**1, 4**)** G.add\_edge**(**2, 5**)** G.add\_edge**(**3, 6**)** G.add\_edge**(**4, 7**)** G.add\_edge**(**5, 8**)** G.add\_edge**(**6, 8**)** G.add\_edge**(**7, 8**)** G.add\_edge**(**9, 8**)** G.add\_edge**(**10, 9**)** print**(**Graph.dict**)**

(Task 14 Basic)

**class Node:  
 def \_\_init\_\_(**self,value**):** self.value **=** value  
  
  
  
**class Graph:** dict**={}** # key(nodeNameVale): value( Edges)  
 keys **= []** # Stores the keys for the dictionary  
 flagged **= []** # checking  
 stack **= []** queue **= []  
  
 def add\_vertex(**self,value**):** Node**(**value**)** # Creates the new Vertex  
  
 **def add\_edge(**self,value,vertices**):** # Adds the edge connecting the vertex\_1 and vertex\_2  
 Graph.dict**[**value**] =** vertices  
  
 **def breath\_first(**self,node **= None**,count**=**1,first **= False):  
 if** node **== None:** # If no starting node passed on to the function it starts with the first added  
 node **=** Graph.keys**[**0**]  
 if** first **== False:** # Initialise the starting node  
 Graph.flagged.append**(**node**)** # Flag the node as visited  
 Graph.queue.insert**(**0,node**)** # Insert it in a queue( First in / Fist out)  
 count **=** 1  
 first **= True** node **=** Graph.queue.pop**()** # Pop the current node  
 f **=** open**('traversal'**, **'a+')** # Printing it in a text file  
 f.write**("BFS: " +** str**(**node**) + "\n")** f.close**()** print**(**node**)  
 for** edge **in** Graph.dict**[**node**]:** # For every edge of the current node, flag it  
 **if** edge **not in** Graph.flagged**:** Graph.flagged.append**(**edge**)  
 try:** Graph.queue.append**(**Graph.flagged**[**count**])** # Add the next flagged node to the queue  
 self.breath\_first**(**node,count**+**1,first**)** # Call the function updating the count which points to the next flagged node to become current node  
 **except:** # Function stops the recursion when a run-time error occurs ( all items iterated - no more flagged nodes to add)  
 **pass  
  
  
 def depth\_first(**self,node**=None**,first **= False):  
 if** node **== None:** # If the starting node isn't provided take the first one added  
 node **=** Graph.keys**[**0**]  
 if** first **== False:** # Initialising the first node  
 Graph.flagged.append**(**node**)** Graph.stack.append**(**node**)** first **= True** count\_edge **=** len**(**Graph.dict**[**node**]) -** 1 # Counts down the visited edges so we now when all of them have been visited  
 **for** edge **in** Graph.dict**[**node**]:** # Goes down on every edge of this node  
 count\_edge **-=** 1  
 **if** edge **not in** Graph.flagged**:** # If the edge is not visited, mark it as such and add it to the stack. Then recourse the function with it.  
 Graph.stack.append**(**edge**)** Graph.flagged.append**(**edge**)** self.depth\_first**(**edge,first**)  
 if** count\_edge **<** 0**:** # If the current node has no more edges that haven't been visited, it gets popped and printed.  
 a **=** Graph.stack.pop**()** f **=** open**('traversal'**, **'a+')** f.write**("DFS : " +** str**(**a**) + "\n")** f.close**()** print **(**a**)  
  
if** \_\_name\_\_ **== '\_\_main\_\_':** G **=** Graph**()** G.insert**(**1, **[**2,3,4**])** G.insert**(**2, **[**1,5**])** G.insert**(**3, **[**1,6**])** G.insert**(**4, **[**1,7**])** G.insert**(**5, **[**2,8**])** G.insert**(**6, **[**3,8**])** G.insert**(**7, **[**4,8**])** G.insert**(**8, **[**5,6,7,9**])** G.insert**(**9, **[**8,10**])** G.insert**(**10, **[**9**])** #G.depth\_first()  
 G.breath\_first**()**

Week 8

(Task 15 Basic) \_\_\_\_\_NOT FINNISHED

**class Node:  
 def \_\_init\_\_(**self,value**):** self.value **=** value  
  
**class Graph:** dict **= {}** # key(nodeNameVale): value( Edges) //edges added here are for the ease of the use on BFS and DFS  
 keys **= []** # Stores the keys for the dictionary  
 edge\_matrix **= []** # 2D list storing the nodes edges  
 edge\_position **= {}** # used to relate the edges to the nodes  
  
 flagged **= []** stack **= []** queue **= []  
  
  
 def add\_vertex(**self,value**):** Node**(**value**)** # Creates the new Vertex  
 self.dict**[**value**] = []** self.keys.append**(**value**)  
 for** row **in** self.edge\_matrix**:** # Increases the length of all the previous rows of edges when inserting a new one  
 row.append**(-**1**)** self.edge\_matrix.append**([-**1**] \* (**len**(**self.edge\_matrix**) +** 1**) )** # Adds the new vertex's(edges) with the size of the previous ones  
 sth **=** str**(**value**)** self.edge\_position**[**sth**] =** len**(**self.edge\_position**)** # Stores the position of the list of edges related to the vertex  
  
 **def add\_edge(**self,vertex\_1,vertex\_2,weight**):** # Adds the edge connecting the vertex\_1 and vertex\_2  
 self.edge\_matrix**[**self.edge\_position**[**str**(**vertex\_1**)]][**self.edge\_position**[**str**(**vertex\_2**)]] =** weight  
 self.edge\_matrix**[**self.edge\_position**[**str**(**vertex\_2**)]][**self.edge\_position**[**str**(**vertex\_1**)]] =** weight  
 self.dict**[**vertex\_1**]**.append**(**vertex\_2**)** self.dict**[**vertex\_2**]**.append**(**vertex\_1**)  
  
  
  
 def breath\_first(**self,node **= None**,count**=**1,first **= False):  
  
 if** node **== None:** # If no starting node passed on to the function it starts with the first added  
 node **=** Graph.keys**[**0**]  
 if** first **== False:** # Initialise the starting node  
 Graph.flagged.append**(**node**)** # Flag the node as visited  
 Graph.queue.insert**(**0,node**)** # Insert it in a queue( First in / Fist out)  
 count **=** 1  
 first **= True** node **=** Graph.queue.pop**()** # Pop the current node  
 f **=** open**('traversal'**, **'a+')** # Printing it in a text file  
 f.write**("BFS: " +** str**(**node**) + "\n")** f.close**()  
 for** edge **in** Graph.dict**[**node**]:** # For every edge of the current node, flag it  
 **if** edge **not in** Graph.flagged**:** Graph.flagged.append**(**edge**)  
 try:** Graph.queue.append**(**Graph.flagged**[**count**])** # Add the next flagged node to the queue  
 self.breath\_first**(**node,count**+**1,first**)** # Call the function updating the count which points to the next flagged node to become current node  
 **except:** # Function stops the recursion when a run-time error occurs ( all items iterated - no more flagged nodes to add)  
 **pass  
  
 def depth\_first(**self,node**=None**,first **= False):  
 if** node **== None:** # If the starting node isn't provided take the first one added  
 node **=** Graph.keys**[**0**]  
 if** first **== False:** # Initialising the first node  
 Graph.flagged.append**(**node**)** Graph.stack.append**(**node**)** first **= True** count\_edge **=** len**(**Graph.dict**[**node**]) -** 1 # Counts down the visited edges so we now when all of them have been visited  
 **for** edge **in** Graph.dict**[**node**]:** # Goes down on every edge of this node  
 count\_edge **-=** 1  
 **if** edge **not in** Graph.flagged**:** # If the edge is not visited, mark it as such and add it to the stack. Then recourse the function with it.  
 Graph.stack.append**(**edge**)** Graph.flagged.append**(**edge**)** self.depth\_first**(**edge,first**)  
 if** count\_edge **<** 0**:** # If the current node has no more edges that haven't been visited, it gets popped and printed.  
 a **=** Graph.stack.pop**()** f **=** open**('traversal'**, **'a+')** f.write**("DFS : " +** str**(**a**) + "\n")** f.close**()  
  
  
  
 def Dijkstra(**self,start,end, all\_vertex **=** keys, vertex\_edges **=** edge\_matrix**):** table **= [[]**,**[]**,**[]**,**[]]** # Table containing : Vertex | Known | Cost | Path |  
 visited **= []** queue **= []** vertex\_paths **= []** current\_vertex **=** start  
  
 **for** vertex **in** all\_vertex**:** # Fill in the table  
 table**[**0**]**.append**(**vertex**)** table**[**1**]**.append**(False)** table**[**2**]**.append**(**float**("inf"))** table**[**3**]**.append**(-**1**)** table**[**2**][**0**] =** 0  
  
 visited.append**(**current\_vertex**)** queue.insert**(**0,current\_vertex**)  
 while False in** table**[**1**]:** table**[**1**][**all\_vertex.index**(**current\_vertex**)] = True** # Finalize the vertex  
 #print (table) #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
 **for** edge **in** vertex\_edges**[**all\_vertex.index**(**current\_vertex**)]:** print **(**all\_vertex**[**vertex\_edges**[**all\_vertex.index**(**current\_vertex**)]**.index**(**edge**)])** print **(**table**)** print **(**queue**)  
  
 if** edge **!= -**1 **and** all\_vertex**[**vertex\_edges**[**all\_vertex.index**(**current\_vertex**)]**.index**(**edge**)] not in** visited**:** #print(all\_vertex.index(vertex\_edges.index(edge)))  
 visited.append**(**all\_vertex**[**vertex\_edges**[**all\_vertex.index**(**current\_vertex**)]**.index**(**edge**)])** queue.insert**(**0, all\_vertex**[**vertex\_edges**[**all\_vertex.index**(**current\_vertex**)]**.index**(**edge**)])** #print (queue) #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
 **if** table**[**2**][**all\_vertex.index**(**edge**)] >** table**[**2**][**all\_vertex.index**(**current\_vertex**)] +** vertex\_edges**[**current\_vertex**][**edge**]:** # If (tentative weight of the vertex) > (current\_vertex(cost) + edge(cost)  
 table**[**2**][**all\_vertex.index**(**edge**)] =** vertex\_edges**[**current\_vertex**][**edge**]  
 else:  
 continue** queue.pop**()** current\_vertex **=** queue**[**0**]  
  
  
if** \_\_name\_\_ **== '\_\_main\_\_':** G **=** Graph**()** G.add\_vertex**(**1**)** G.add\_vertex**(**2**)** G.add\_vertex**(**3**)** G.add\_vertex**(**4**)** G.add\_vertex**(**5**)** G.add\_vertex**(**6**)** G.add\_vertex**(**7**)** G.add\_vertex**(**8**)** G.add\_vertex**(**9**)** G.add\_vertex**(**10**)** G.add\_edge**(**1, 2, 1**)** G.add\_edge**(**1, 3, 2**)** G.add\_edge**(**1, 4, 3**)** G.add\_edge**(**2, 5, 4**)** G.add\_edge**(**3, 6, 5**)** G.add\_edge**(**4, 7, 2**)** G.add\_edge**(**5, 8, 1**)** G.add\_edge**(**6, 8, 4**)** G.add\_edge**(**7, 8, 6**)** G.add\_edge**(**9, 8, 7**)** G.add\_edge**(**10, 9, 3**)** # print(Graph.edge\_matrix)  
  
  
 #G.breath\_first()  
 G.depth\_first**()** # G.Dijkstra(1,9)

GITHUB LINK : https://github.com/Chris9606/My\_work