Data Structures:

1. BST (Preorder, Postorder and Inorder)
2. Linked List/ Doubly Linked List
3. Fenwick Tree/ Segment Tree
4. Defaultdict, Heapq, Bisect
5. Graph template
6. Trie

Graph Algorithms:

1. BFS/ DFS
2. Prim’s
3. Kruskal’s
4. Dijkstra
5. Strongly connected components/ Connected components
6. In Degree / Outdegree
7. Top Sort

Strings:

1. KMP/ Rabin-Karp

DP:

1. Longest increasing subsequence/subarray
2. Knapsack problem
3. Maximum sum rectangle in a 2D Matrix

Greedy:

Maths:

1. GCD
2. Sieve of Eratosthenes
3. nCr
4. Pascal triangle
5. Euler Totient
6. Chinese remainder theorem
7. Euler Phi Function
8. Modular Arithmetic
9. XOR Arithmetic
10. AND, OR, NOT
11. Fast Fibonacci
12. Golden ratio

**Data Structures:**

1. **Binary Search Tree** –

class Node(object):

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

self.data = d

self.left = None

self.right = None

def insert(self, d):

if self.data == d:

return False

elif d < self.data:

if self.left:

return self.left.insert(d)

else:

self.left = Node(d)

return True

else:

if self.right:

return self.right.insert(d)

else:

self.right = Node(d)

return True

def find(self, d):

if self.data == d:

return True

elif d < self.data and self.left:

return self.left.find(d)

elif d > self.data and self.right:

return self.right.find(d)

return False

def preorder(self, l):

l.append(self.data)

if self.left:

self.left.preorder(l)

if self.right:

self.right.preorder(l)

return l

def postorder(self, l):

if self.left:

self.left.postorder(l)

if self.right:

self.right.postorder(l)

l.append(self.data)

return l

def inorder(self, l):

if self.left:

self.left.inorder(l)

l.append(self.data)

if self.right:

self.right.inorder(l)

return l

class BST(object):

  def \_\_init\_\_(self):

    self.root = None

  def insert(self, d):

    if self.root:

      return self.root.insert(d)

    else:

      self.root = Node(d)

      return True

  def find(self, d):

    if self.root:

      return self.root.find(d)

    else:

      return False

  def remove(self, d):

    if self.root == None:

      return False

    if self.root.data == d:

      if self.root.left is None and self.root.right is None:

        self.root = None

        return True

      elif self.root.left and self.root.right is None:

        self.root = self.root.left

        return True

      elif self.root.left is None and self.root.right:

        self.root = self.root.right

        return True

      else:

        moveNode = self.root.right

        moveNodeParent = None

        while moveNode.left:

          moveNodeParent = moveNode

          moveNode = moveNode.left

        self.root.data = moveNode.data

        if moveNode.data < moveNodeParent.data:

          moveNodeParent.left = None

        else:

          moveNodeParent.right = None

        return True

    parent = None

    node = self.root

    while node and node.data != d:

      parent = node

      if d < node.data:

        node = node.left

      elif d > node.data:

        node = node.right

    if node == None or node.data != d:

      return False

    elif node.left is None and node.right is None:

      if d < parent.data:

        parent.left = None

      else:

        parent.right = None

      return True

    elif node.left and node.right is None:

      if d < parent.data:

        parent.left = node.left

      else:

        parent.right = node.left

      return True

    elif node.left is None and node.right:

      if d < parent.data:

        parent.left = node.right

      else:

        parent.right = node.right

      return True

    else:

      moveNodeParent = node

      moveNode = node.right

      while moveNode.left:

        moveNodeParent = moveNode

        moveNode = moveNode.left

      node.data = moveNode.data

      if moveNode.right:

        if moveNode.data < moveNodeParent.data:

          moveNodeParent.left = moveNode.right

        else:

          moveNodeParent.right = moveNode.right

      else:

        if moveNode.data < moveNodeParent.data:

          moveNodeParent.left = None

        else:

          moveNodeParent.right = None

      return True

  def preorder(self):

    if self.root:

      return self.root.preorder([])

    else:

      return []

  def postorder(self):

    if self.root:

      return self.root.postorder([])

    else:

      return []

  def inorder(self):

    if self.root:

      return self.root.inorder([])

    else:

      return []

1. **Linked List (Insert and delete)** –

class Node:

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

self.data = data

self.next = None

class LinkedList:

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

self.head = None

def printList(self):

temp = self.head

while (temp):

print (temp.data,)

temp = temp.next

def insertFirst(self, new\_data):

new\_node = Node(new\_data)

new\_node.next = self.head

self.head = new\_node

def insertAfter(self, prev\_node, new\_data):

if prev\_node is None:

print "The given previous node must in LinkedList."

return

new\_node = Node(new\_data)

new\_node.next = prev\_node.next

prev\_node.next = new\_node

def append(self, new\_data):

new\_node = Node(new\_data)

if self.head is None:

self.head = new\_node

return

last = self.head

while (last.next):

last = last.next

last.next = new\_node

def deleteNode(self, key):

temp = self.head

if (temp is not None):

if (temp.data == key):

self.head = temp.next

temp = None

return

while(temp is not None):

if temp.data == key:

break

prev = temp

temp = temp.next

if(temp == None):

return

prev.next = temp.next

temp = None

1. **Fenwick Tree** –

def getsum(BITTree,i):

i = i+1

while i > 0:

s += BITTree[i]

i -= i & (-i)

return s

def updatebit(BITTree , n , i ,v):

i += 1

while i <= n:

BITTree[i] += v

i += i & (-i)

def construct(arr, n):

BITTree = [0]\*(n+1)

for i in range(n):

updatebit(BITTree, n, i, arr[i])

return BITTree

1. **Graph and functions** –

class Graph(object):

def \_\_init\_\_(self, graph\_dict=None):

#initializes a graph object

if graph\_dict == None:

graph\_dict = {}

self.\_\_graph\_dict = graph\_dict

def vertices(self):

""" returns the vertices of a graph """

return list(self.\_\_graph\_dict.keys())

def edges(self):

""" returns the edges of a graph """

return self.\_\_generate\_edges()

def add\_vertex(self, vertex):

"""If the vertex "vertex" is not in

self.\_\_graph\_dict, a key "vertex" with an empty

list as a value is added to the dictionary.

Otherwise nothing has to be done.

"""

if vertex not in self.\_\_graph\_dict:

self.\_\_graph\_dict[vertex] = []

def add\_edge(self, edge):

"""assumes that edge is of type set, tuple or list

between two vertices can be multiple edges!

"""

edge = set(edge)

vertex1 = edge.pop()

if edge:

# not a loop

vertex2 = edge.pop()

else:

# a loop

vertex2 = vertex1

if vertex1 in self.\_\_graph\_dict:

self.\_\_graph\_dict[vertex1].append(vertex2)

else:

self.\_\_graph\_dict[vertex1] = [vertex2]

def \_\_generate\_edges(self):

""" A static method generating the edges of the

graph "graph". Edges are represented as sets

with one (a loop back to the vertex) or two

vertices """

edges = []

for vertex in self.\_\_graph\_dict:

for neighbour in self.\_\_graph\_dict[vertex]:

if {neighbour, vertex} not in edges:

edges.append({vertex, neighbour})

return edges

def \_\_str\_\_(self):

res = "vertices: "

for k in self.\_\_graph\_dict:

res += str(k) + " "

res += "\nedges: "

for edge in self.\_\_generate\_edges():

res += str(edge) + " "

return res

def find\_isolated\_vertices(self):

""" returns a list of isolated vertices. """

graph = self.\_\_graph\_dict

isolated = []

for vertex in graph:

print(isolated, vertex)

if not graph[vertex]:

isolated += [vertex]

return isolated

def find\_path(self, start\_vertex, end\_vertex, path=[]):

"""find a path from start\_vertex to end\_vertex

in graph """

graph = self.\_\_graph\_dict

path = path + [start\_vertex]

if start\_vertex == end\_vertex:

return path

if start\_vertex not in graph:

return None

for vertex in graph[start\_vertex]:

if vertex not in path:

extended\_path = self.find\_path(vertex,

end\_vertex,

path)

if extended\_path:

return extended\_path

return None

def find\_all\_paths(self, start\_vertex, end\_vertex, path=[]):

"""ind all paths from start\_vertex to

end\_vertex in graph """

graph = self.\_\_graph\_dict

path = path + [start\_vertex]

if start\_vertex == end\_vertex:

return [path]

if start\_vertex not in graph:

return []

paths = []

for vertex in graph[start\_vertex]:

if vertex not in path:

extended\_paths = self.find\_all\_paths(vertex,

end\_vertex,

path)

for p in extended\_paths:

paths.append(p)

return paths

def is\_connected(self,

vertices\_encountered = None,

start\_vertex=None):

""" determines if the graph is connected """

if vertices\_encountered is None:

vertices\_encountered = set()

gdict = self.\_\_graph\_dict

vertices = list(gdict.keys()) # "list" necessary in Python 3

if not start\_vertex:

# chosse a vertex from graph as a starting point

start\_vertex = vertices[0]

vertices\_encountered.add(start\_vertex)

if len(vertices\_encountered) != len(vertices):

for vertex in gdict[start\_vertex]:

if vertex not in vertices\_encountered:

if self.is\_connected(vertices\_encountered, vertex):

return True

else:

return True

return False

def vertex\_degree(self, vertex):

""" The degree of a vertex is the number of edges connecting

it, i.e. the number of adjacent vertices. Loops are counted

double, i.e. every occurence of vertex in the list

of adjacent vertices. """

adj\_vertices = self.\_\_graph\_dict[vertex]

degree = len(adj\_vertices) + adj\_vertices.count(vertex)

return degree

def degree\_sequence(self):

""" calculates the degree sequence """

seq = []

for vertex in self.\_\_graph\_dict:

seq.append(self.vertex\_degree(vertex))

seq.sort(reverse=True)

return tuple(seq)

@staticmethod

def is\_degree\_sequence(sequence):

""" Method returns True, if the sequence "sequence" is a

degree sequence, i.e. a non-increasing sequence.

Otherwise False is returned.

"""

# check if the sequence sequence is non-increasing:

return all( x>=y for x, y in zip(sequence, sequence[1:]))

def delta(self):

""" the minimum degree of the vertices """

min = 100000000

for vertex in self.\_\_graph\_dict:

vertex\_degree = self.vertex\_degree(vertex)

if vertex\_degree < min:

min = vertex\_degree

return min

def Delta(self):

""" the maximum degree of the vertices """

max = 0

for vertex in self.\_\_graph\_dict:

vertex\_degree = self.vertex\_degree(vertex)

if vertex\_degree > max:

max = vertex\_degree

return max

def density(self):

""" method to calculate the density of a graph """

g = self.\_\_graph\_dict

V = len(g.keys())

E = len(self.edges())

return 2.0 \* E / (V \*(V - 1))

def diameter(self):

""" calculates the diameter of the graph """

v = self.vertices()

pairs = [ (v[i],v[j]) for i in range(len(v)) for j in range(i+1, len(v)-1)]

smallest\_paths = []

for (s,e) in pairs:

paths = self.find\_all\_paths(s,e)

smallest = sorted(paths, key=len)[0]

smallest\_paths.append(smallest)

smallest\_paths.sort(key=len)

diameter = len(smallest\_paths[-1]) - 1

return diameter

1. **Trie** –

class TrieNode:

# Trie node class

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

self.children = [None]\*26

#EndofWord condition check

self.isEndOfWord = False

class Trie:

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

self.root = self.getNode()

def getNode(self):

# Returns new trie node (initialized to NULLs)

return TrieNode()

def \_charToIndex(self,ch):

# private helper function

# Converts key current character into index

# use only 'a' through 'z' and lower case

return ord(ch)-ord('a')

def insert(self,key):

# If not present, inserts key into trie

# If the key is prefix of trie node,

# just marks leaf node

pCrawl = self.root

length = len(key)

for level in range(length):

index = self.\_charToIndex(key[level])

# if current character is not present

if not pCrawl.children[index]:

pCrawl.children[index] = self.getNode()

pCrawl = pCrawl.children[index]

# mark last node as leaf

pCrawl.isEndOfWord = True

def search(self, key):

# Search key in the trie

# Returns true if key presents

# in trie, else false

pCrawl = self.root

length = len(key)

for level in range(length):

index = self.\_charToIndex(key[level])

if not pCrawl.children[index]:

return False

pCrawl = pCrawl.children[index]

return (pCrawl != None and pCrawl.isEndOfWord)

**String algorithms:**

1. **KMP Algorithm** –

def KMPSearch(pat, txt):

M = len(pat)

N = len(txt)

# create lps[] that will hold the longest prefix suffix

# values for pattern

lps = [0]\*M

j = 0 # index for pat[]

# Preprocess the pattern (calculate lps[] array)

computeLPSArray(pat, M, lps)

i = 0 # index for txt[]

while i < N:

if pat[j] == txt[i]:

i += 1

j += 1

if j == M:

print "Found pattern at index " + str(i-j)

j = lps[j-1]

# mismatch after j matches

elif i < N and pat[j] != txt[i]:

# Do not match lps[0..lps[j-1]] characters,

# they will match anyway

if j != 0:

j = lps[j-1]

else:

i += 1

def computeLPSArray(pat, M, lps):

len = 0

lps[0]

i = 1

# the loop calculates lps[i] for i = 1 to M-1

while i < M:

if pat[i]== pat[len]:

len += 1

lps[i] = len

i += 1

else:

if len != 0:

len = lps[len-1]

else:

lps[i] = 0

i += 1

**Graph algorithms:**

1. **BFS and DFS** –

def Graph():

  graph={}

  return(graph)

def addPath(n1,n2,graph):

  try:

    graph[n1].append(n2)

  except:

    graph[n1]=[n2]

  #Uncomment Below lines if Graph is Undirected

  '''try:

    graph[n2].append(n1)

  except:

    graph[n2]=[n1]'''

  return(graph)

def bfs(graph,start):

  n=len(graph)

  visited=[False]\*n

  queue=[start]

  ans=[]

  while(queue!=[]):

    node=queue.pop(0)

    ans.append(node)

    visited[node]=True

    for i in graph[node]:

      if(visited[i]==False):

        queue.append(i)

  return(ans)

def dfs(graph,start):

  n=len(graph)

  visited=[False]\*n

  stack=[start]

  ans=[]

  while(stack!=[]):

    node=stack.pop()

    ans.append(node)

    visited[node]=True

    for i in graph[node]:

      if(visited[i]==False):

        stack.append(i)

  return(ans)

1. **Prim’s Algorithm (Adjacency Matrix)** -

import sys

class Graph():

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

self.V = vertices

self.graph = [[0 for column in range(vertices)]

for row in range(vertices)]

def printMST(self, parent):

print "Edge \tWeight"

for i in range(1, self.V):

print parent[i], "-", i, "\t", self.graph[i][ parent[i] ]

# A utility function to find the vertex with

# minimum distance value, from the set of vertices

# not yet included in shortest path tree

def minKey(self, key, mstSet):

# Initilaize min value

min = sys.maxint

for v in range(self.V):

if key[v] < min and mstSet[v] == False:

min = key[v]

min\_index = v

return min\_index

def primMST(self):

key = [sys.maxint] \* self.V

parent = [None] \* self.V

key[0] = 0

mstSet = [False] \* self.V

parent[0] = -1

for cout in range(self.V):

# Pick the minimum distance vertex from

# the set of vertices not yet processed.

u = self.minKey(key, mstSet)

# Put the minimum distance vertex in

# the shortest path tree

mstSet[u] = True

# Update dist value of the adjacent vertices

# of the picked vertex only if the current

# distance is greater than new distance and

# the vertex in not in the shotest path tree

for v in range(self.V):

if self.graph[u][v] > 0 and mstSet[v] == False and key[v] > self.graph[u][v]:

key[v] = self.graph[u][v]

parent[v] = u

self.printMST(parent)

1. **Kruskal’s Algorithm (Dictionary/List)** –

import sys

class Graph():

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

self.V = vertices

self.graph = [[0 for column in range(vertices)]

for row in range(vertices)]

def printMST(self, parent):

print "Edge \tWeight"

for i in range(1, self.V):

print parent[i], "-", i, "\t", self.graph[i][ parent[i] ]

# A utility function to find the vertex with

# minimum distance value, from the set of vertices

# not yet included in shortest path tree

def minKey(self, key, mstSet):

# Initilaize min value

min = sys.maxint

for v in range(self.V):

if key[v] < min and mstSet[v] == False:

min = key[v]

min\_index = v

return min\_index

# Function to construct and print MST for a graph

# represented using adjacency matrix representation

def primMST(self):

key = [sys.maxint] \* self.V

parent = [None] \* self.V

key[0] = 0

mstSet = [False] \* self.V

parent[0] = -1

for cout in range(self.V):

# Pick the minimum distance vertex from

# the set of vertices not yet processed.

u = self.minKey(key, mstSet)

# Put the minimum distance vertex in

# the shortest path tree

mstSet[u] = True

# Update dist value of the adjacent vertices

# of the picked vertex only if the current

# distance is greater than new distance and

# the vertex in not in the shotest path tree

for v in range(self.V):

if self.graph[u][v] > 0 and mstSet[v] == False and key[v] > self.graph[u][v]:

key[v] = self.graph[u][v]

parent[v] = u

self.printMST(parent)

1. **Dijkstra’s Algorithm (Adjacency Matrix)** –

import sys

class Graph():

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

self.V = vertices

self.graph = [[0 for column in range(vertices)]

for row in range(vertices)]

def printSolution(self, dist):

print "Vertex \tDistance from Source"

for node in range(self.V):

print node, "\t", dist[node]

# A utility function to find the vertex with

# minimum distance value, from the set of vertices

# not yet included in shortest path tree

def minDistance(self, dist, sptSet):

# Initialize minimum distance for next node

min = sys.maxint

# Search not nearest vertex not in the

# shortest path tree

for v in range(self.V):

if dist[v] < min and sptSet[v] == False:

min = dist[v]

min\_index = v

return min\_index

# Function that implements Dijkstra's single source

# shortest path algorithm for a graph represented

# using adjacency matrix representation

def dijkstra(self, src):

dist = [sys.maxint] \* self.V

dist[src] = 0

sptSet = [False] \* self.V

for cout in range(self.V):

# Pick the minimum distance vertex from

# the set of vertices not yet processed.

# u is always equal to src in first iteration

u = self.minDistance(dist, sptSet)

# Put the minimum distance vertex in the

# shotest path tree

sptSet[u] = True

# Update dist value of the adjacent vertices

# of the picked vertex only if the current

# distance is greater than new distance and

# the vertex in not in the shotest path tree

for v in range(self.V):

if (self.graph[u][v] > 0 and sptSet[v] == False and dist[v] > dist[u] + self.graph[u][v]):

dist[v] = dist[u] + self.graph[u][v]

self.printSolution(dist)

1. **Strongly Connected components (Adjacency List) –**

from collections import defaultdict

class Graph:

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

self.V= vertices

self.graph = defaultdict(list)

def addEdge(self,u,v):

self.graph[u].append(v)

# A function used by DFS

def DFSUtil(self,v,visited):

for i in self.graph[v]:

if visited[i]==False:

self.DFSUtil(i,visited)

def fillOrder(self,v,visited, stack):

for i in self.graph[v]:

if visited[i]==False:

self.fillOrder(i, visited, stack)

stack = stack.append(v)

# Function that returns reverse (or transpose) of this graph

def getTranspose(self):

g = Graph(self.V)

for i in self.graph:

for j in self.graph[i]:

g.addEdge(j,i)

return g

# The main function that finds and prints all strongly connected components

def printSCCs(self):

stack = []

# Mark all the vertices as not visited

visited =[False]\*(self.V)

for i in range(self.V):

if visited[i]==False:

self.fillOrder(i, visited, stack)

gr = self.getTranspose()

visited =[False]\*(self.V)

while stack:

i = stack.pop()

if visited[i]==False:

gr.DFSUtil(i, visited)

print ("")

1. **Connected components of a graph (Adjacency List) –**

class Graph:

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

self.V = V

self.adj = [[] for i in range(V)]

def DFSUtil(self, temp, v, visited):

visited[v] = True

temp.append(v)

for i in self.adj[v]:

if visited[i] == False:

temp = self.DFSUtil(temp, i, visited)

return temp

# method to add an undirected edge

def addEdge(self, v, w):

self.adj[v].append(w)

self.adj[w].append(v)

def connectedComponents(self):

visited = []

cc = []

for i in range(self.V):

visited.append(False)

for v in range(self.V):

if visited[v] == False:

temp = []

cc.append(self.DFSUtil(temp, v, visited))

return cc

1. **Finding indegree and outdegree of all vertices in a graph (Adjacency List) –**

def findInOutDegree(adjList, n):

\_in = [0] \* n

out = [0] \* n

for i in range(0, len(adjList)):

List = adjList[i]

# Out degree for ith vertex will be the count

# of direct paths from i to other vertices

out[i] = len(List)

for j in range(0, len(List)):

# Every vertex that has

# an incoming edge from i

\_in[List[j]] += 1

print("Vertex\tIn\tOut")

for k in range(0, n):

print(str(k) + "\t" + str(\_in[k]) +

"\t" + str(out[k]))

1. **Topological Sort (Adjacency List) –**

from collections import defaultdict

class Graph:

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

self.graph = defaultdict(list)

self.V = vertices

def addEdge(self,u,v):

self.graph[u].append(v)

def topologicalSortUtil(self,v,visited,stack):

visited[v] = True

for i in self.graph[v]:

if visited[i] == False:

self.topologicalSortUtil(i,visited,stack)

stack.insert(0,v)

def topologicalSort(self):

visited = [False]\*self.V

stack =[]

for i in range(self.V):

if visited[i] == False:

self.topologicalSortUtil(i,visited,stack)

print (stack)

**Dynamic Programming:**

1. **Longest increasing subsequence –**

def lis(arr):

n = len(arr)

lis = [1]\*n

for i in range (1 , n):

for j in range(0 , i):

if arr[i] > arr[j] and lis[i]< lis[j] + 1 :

lis[i] = lis[j]+1

maximum = 0

for i in range(n):

maximum = max(maximum , lis[i])

return maximum

1. **Longest increasing subarray –**

def printLogestIncSubArr( arr, n) :

m = 1

l = 1

maxIndex = 0

for i in range(1, n) :

if (arr[i] > arr[i-1]) :

l =l + 1

else :

if (m < l) :

m = l

maxIndex = i - m

l = 1

if (m < l) :

m = l

maxIndex = n - m

for i in range(maxIndex, (m+maxIndex)) :

print(arr[i] , end=" ")

1. **Knapsack problem –**