I have a python task for you to implemented Kruskal's algorithm for finding a Minimum Spanning Tree.
I will provide two py files, one is GA_ProjectUtils.py which should not modify, one is mst.py which the one you should modify.
Because the whole file is too long, I will separate them and send to you

file GA_ProjectUtils.py
file mst.py

```
"""
    mst.py       Intro to Graduate Algorithms, Summer 2023

    You will implement Kruskal's algorithm for finding a Minimum Spanning Tree.
    You will also implement the union-find data structure using path compression

    Only modify this template where instructed.
    Do not change the signatures of any functions.
    Do not import any libraries beyond those included by the template.
    You must complete every function

    Notes on data structures:
        vertex IDs are represented as integers in the range 0...(n-1)
        edges are represented as (u,v) tuples where u & v are vertex IDs
        n or G.numVerts represents the number of vertices in the graph G
"""

import argparse
import GA_ProjectUtils as util

class unionFind:
    def __init__(self, n):
        self.pi = [i for i in range(n)]
        self.rank = [0 for i in range(n)]

    def union(self, u, v):
        """
            u & v are two vertices, each is in a different component
            build union of 2 components
            Be sure to maintain self.rank as needed to
            make sure your algorithm is optimal.
        """
        #TODO Your Code Goes Here

    def find(self, p):
        """
            find the root of the set containing the
            passed vertex p - Must use path compression!
        """
        #TODO Your Code Goes Here

def kruskal(G):
    """
        Kruskal algorithm
        G : graph object
    """
    #Build unionFind Object
    uf = unionFind(G.numVerts)
    #Make MST as a set
    MST = set()
    #Get list of edges sorted by increasing weight
    sortedEdges = G.sortedEdges()
    #Go through edges in sorted order smallest, to largest
    for e in sortedEdges:
        #TODO Your Code Goes Here (remove comments if you wish)

        # use the following line to add an edge to the MST.
        # You may change it's indentation/scope within the code
        # but do not otherwise modify it

        MST.add(util.buildMSTEdge(G,e))

        #Done - do not modify any other code below this line
    return MST, uf

```

```
72  def main():
73      """
74      main
75      """
76      #DO NOT REMOVE ANY ARGUMENTS FROM THE ARGPARSER BELOW
77      parser = argparse.ArgumentParser(description='Minimum Spanning Tree')
78      #use this flag, or change the default here to use different graph description
        files
79      parser.add_argument('-g', '--graphFile',  help='File holding graph data in
        specified format', default='small.txt', dest='graphDataFileName')
80      #use this flag to print the graph and resulting MST
81      parser.add_argument('-p', '--print', help='Print the MSTs?', default=False,
        dest='printMST')
82      parser.add_argument('-pg', '--print-graph', help='Print the graph?',
        default=False, dest='printGRAPH')
83      #args for autograder, DO NOT MODIFY ANY OF THESE
84      parser.add_argument('-a', '--autograde',  help='Autograder-called (2) or not
        (1=default)', type=int, choices=[1, 2], default=1, dest='autograde')
85      args = parser.parse_args()
86
87      #DO NOT MODIFY ANY OF THE FOLLOWING CODE
88      #Build graph object
89      graph = util.build_MSTBaseGraph(args)
90      #you may print the configuration of the graph -- only effective for graphs with
        up to 10 vertex
91      if args.printGRAPH: graph.printMe()
92
93      #Calculate kruskal's alg for MST
94      MST_Kruskal, uf = kruskal(graph)
95
96      #verify against provided prim's algorithm results
97      util.verify_MSTKruskalResults(args, MST_Kruskal, args.printMST)
98
99  if __name__ == '__main__':
100     main()
101
102 """
103 Utility functions - do not modify these functions!  Some of these functions may not
    be applicable to your project.  Ignore them
104
105 If you find errors post to class page.
106
107 """
108 #import time
109 #import os
110 #useful structure to build dictionaries of lists
111 #from collections import defaultdict
112
113 ########################################
114 #IO and Util functions
115
116 #returns sorted version of l, and idx order of sort
117 def getSortResIDXs(l, rev=True):
118     from operator import itemgetter
119     return list(zip(*sorted([(i,e) for i,e in enumerate(l)],
120                             key=itemgetter(1),reverse=rev)))
121
122
123 #read srcFile into list of ints
124 def readIntFileDat(srcFile):
125     strs = readFileDat(srcFile)
126     res = [int(s.strip()) for s in strs]
127     return res
128
129 #read srcFile into list of floats
130 def readFloatFileDat(srcFile):
131     strs = readFileDat(srcFile)
132     res = [float(s.strip()) for s in strs]
133     return res
134
135 #read srcFile into list of strings
136 def readFileDat(srcFile):
137     import os
```

```
138       try:
139           f = open(srcFile, 'r')
140       except IOError:
141           #file doesn't exist, return empty list
142           print(('Note : {} does not exist in current dir : {}'.format(srcFile,
          os.getcwd())))
143           return []
144       src_lines = f.readlines()
145       f.close()
146       return src_lines
147
148   #write datList into fName file
149   def writeFileDat(fName, datList):
150       f = open(fName, 'w')
151       for item in datList:
152           print(item, file=f)
153       f.close()
154
155   #append record to existing file
156   def appendFileDat(fName, dat):
157       f = open(fName, 'a+')
158       print(dat, file=f)
159       f.close()
160
161
162   ########################################
163   #Homework mini-project utility functions
164
165   ##Knapsack
166
167   #this will build a default dictionary of items, where the key is the item number
      (1..n)
168   #and value is tuple of (name, item weight, value)
169
170   def buildKnapsackItemsDict(args):
171       ksItemsData = readFileDat(args.itemsListFileName)
172       items = {}
173       itemCount = 0
174       for line in ksItemsData:
175           itemCount += 1
176           vals = line.split(',')
177           tupleVal = (vals[0].strip(), int(vals[1].strip()), int(vals[2].strip()))
178           items[itemCount] = tupleVal
179
180   #    lst = sorted(res, key = lambda x: x[0])
181
182       if args.autograde == 1:
183           print("The following items were loaded from file {} : \nName, Integer Weight,
          Integer Value : ".format(args.itemsListFileName))
184           for k, val in items.items():
185               print("{0:30} Wt : {1:5} Val : {2:5} ".format(val[0],val[1],val[2]))
186
187       return items
188
189   #Will display results of knapsack problem
190   def displayKnapSack(args, itemsChosen):
191       if(len(itemsChosen)!=0):
192           print("\n\nResults : The following items were chosen : ")
193           lst = sorted(itemsChosen, key = lambda x: x[0])
194           ttlWt = 0
195           ttlVal = 0
196           for s in lst:
197               ttlWt += s[1]
198               ttlVal += s[2]
199               print("{0:30} Wt : {1:5} Val : {2:5} ".format(s[0],s[1],s[2]))
200
201           print(("For a total value of <%i> and a total weight of [%i]" % (ttlVal,
          ttlWt)))
202       else :
203           print("\n\nResults : No Items were chosen: ")
204
205   ##End Knapsack
206
```

```
##MST
#this function will load graph information from file and build the graph structure
def build_MSTBaseGraph(args):
    #file format should be
    #line 0 : # of verts
    #line 1 : # of edges
    #line 2... : vert1 vert2 edgeWT
    MSTGraphData = readFileDat(args.graphDataFileName)
    numVerts = int(MSTGraphData[0].strip())
    numEdges = int(MSTGraphData[1].strip())
    edgeDataAra = []
    for i in range(numEdges):
        line = MSTGraphData[i+2]
        vals = line.split()
        v1 = int(vals[0].strip())
        v2 = int(vals[1].strip())
        wt = float(vals[2].strip())
        #print("v1 :{} v2 :{} wt : {} ".format(v1,v2,wt))
        edgeDataAra.append([wt,v1,v2])

    G = Graph(numVerts, edgeDataAra)
    return G

def print_MSTResults(MST):
    itr = 0
    for E in MST:
        print("({:4d},{:4d}) {:2.6f} ".format(E[1][0], E[1][1], E[0]), end=" | ")
        itr += 1
        if(itr > 2):
            itr=0
            print("")
    print("\n")

"""
    build a tuple holding edge weight and edge verts to add to mst
"""
def buildMSTEdge(G, e):
    wt = G.edgeWts[e]
    return (wt, e)


def save_MSTRes(args, MST):
    saveName = "soln_"+args.graphDataFileName
    strList = []
    for E in MST:
        strDat = "{} {} {}".format(E[1][0],E[1][1],E[0])
        strList.append(strDat)
    writeFileDat(saveName, strList)


def load_MSTRes(args):
    solnName = "soln_"+args.graphDataFileName
    resDataList = readFileDat(solnName)

    MST = set()
    for line in resDataList :
        vals = line.split()
        v1 = int(vals[0].strip())
        v2 = int(vals[1].strip())
        wt = float(vals[2].strip())

        MST.add((wt, (v1,v2)))
    return MST

#u
def findTotalWeightOfMst(MST):
    totWt = 0
    for E in MST:
        totWt += E[0]

    return totWt

#used locally
```

```
280   def _compareTwoMSTs(MST_1, lbl1, MST_2, lbl2, printMST):
281       wt1 = round(findTotalWeightOfMst(MST_1), 12)
282       wt2 = round(findTotalWeightOfMst(MST_2), 12)
283       if(abs(wt1 - wt2) < 1e-12):
284           print("Correct: {} Weight : {} {} Wt : {} ".format(lbl1, wt1, lbl2, wt2))
285           return True
286       else:
287           diff12 = MST_1 - MST_2
288           sizeDiff12 =  len(diff12)
289           diff21 = MST_2 - MST_1
290           sizeDiff21 =  len(diff21)
291           print("Incorrect: {} Weight : {} {} Wt : {}".format(lbl1, wt1, lbl2, wt2))
292           return False
293
294
295   """
296       verifies results of kruskal calculation
297   """
298   def verify_MSTKruskalResults(args, MST_Kruskal, printMST=False):
299       MST_Correct = load_MSTRes(args)
300
301       if(printMST):
302           if(len(MST_Kruskal) < 1):
303               print("No Kruskal's Algorithm results found (Empty MST)")
304           else :
305               print("Kruskal's Algorithm results (Edge list of MST) : ")
306               print_MSTResults(MST_Kruskal)
307           print("\n")
308           print("Correct results : ")
309           print_MSTResults(MST_Correct)
310           print("\n")
311
312       return _compareTwoMSTs(MST_Kruskal,"Kruskal's Result", MST_Correct, "Expected
          Result", printMST)
313
314
315   """
316       this structure will represent an undirected graph as an adjacency matrix
317   """
318   class Graph:
319       def __init__(self, numVerts, edgeDataAra):
320           self.numVerts = numVerts
321           self.numEdges = len(edgeDataAra)
322           self.edgeDataAra = edgeDataAra
323
324           self.edges = set()
325           self.edgeWts = dict()
326
327           # populate the graph
328           for edge in edgeDataAra:
329               #add edge so that lowest vert is always first
330               if(edge[1] > edge[2]):
331                   thisEdge = (edge[2],edge[1])
332               else :
333                   thisEdge = (edge[1],edge[2])
334
335               self.edges.add(thisEdge)
336               self.edgeWts[thisEdge] = edge[0]
337
338       """
339           returns list of edges sorted in increasing weight
340       """
341       def sortedEdges(self):
342           sortedEdges = sorted(self.edges, key=lambda e:self.edgeWts[e])
343           return sortedEdges
344
345       def buildAdjacencyMat(self):
346           numVerts = self.numVerts
347           graphAdjMat = [[0]*numVerts for _ in range(numVerts)]
348           edgeDataAra = self.edgeDataAra
349           for edge in edgeDataAra:
350               graphAdjMat[edge[1]][edge[2]] = edge[0]
351               graphAdjMat[edge[2]][edge[1]] = edge[0]
```

```
352
353         # use adjacent matrix to represent the graph
354         return graphAdjMat
355
356
357     """
358         for debug purposes
359     """
360     def printMe(self):
361         try:
362             print("Graph has :{} vertices and {}
                edges".format(self.numVerts,self.numEdges))
363             NumVerts = min(10, self.numVerts)
364
365             AM = [[0.0 for _ in range(NumVerts)] for _ in range(NumVerts)]
366
367             for edge in self.edges:
368                 a,b = edge
369                 if a > NumVerts: continue
370                 if b > NumVerts: continue
371                 weight = self.edgeWts[edge]
372                 AM[a][b] = weight
373                 AM[b][a] = weight
374
375             print('  ', end = '  ')
376             for i in range(NumVerts):
377                 print('{0:5d}'.format(i), end = '  ')
378             print()
379
380             for i, row in enumerate(AM):
381                 print('{0:2d}'.format(i), end='  ')
382                 for j in row:
383                     if j == 0:
384                         print('     ', end = '  ')
385                     else:
386                         print('{0:1.3f}'.format(j),end='  ')
387                 print()
388
389             print()
390         except:
391             print("Error Rendering Graph...")
392
393
394 ##End MST
395
396 ########################################
397 #Bloom Filter Project functions
398
399 #this will compare the contents of the resList with the data in baseFile
400 #and display performance
401 def compareResults(resList, configData):
402     baseFileName = configData['valFileName']
403     baseRes = readFileDat(baseFileName)
404     if(len(baseRes) != len(resList) ):
405         print('compareFiles : Failure : Attempting to compare different size lists')
406         return None
407     numFail = 0
408     numFTrueRes = 0
409     numFFalseRes = 0
410     for i in range(len(resList)):
411         if (resList[i].strip().lower() != baseRes[i].strip().lower()):
412             resVal = resList[i].strip().lower()
413             baseResVal = baseRes[i].strip().lower()
414             #uncomment this to see inconsistencies
415             #print('i : ' + str(i) + ': reslist : ' + resVal + ' | baseres : ' +
                baseResVal)
416             numFail += 1
417             if resVal == 'true' :
418                 numFTrueRes += 1
419             else :
420                 numFFalseRes += 1
421     if(numFail == 0):
422         print('compareResults : Your bloom filter performs as expected')
```

```
423         else:
424             print(('compareResults : Number of mismatches in bloomfilter compared to
                validation file : ' + str(numFail) + '| # of incorrect true results : ' +
                str(numFTrueRes) + '| # of incorrect False results : ' + str(numFFalseRes)))
425         if((configData['studentName'] != '') and (configData['autograde'] == 2)):
426             gradeRes = configData['studentName'] + ', ' + str(numFail) + ', ' +
                str(numFTrueRes) + ', ' + str(numFFalseRes)
427             print(('saving results for ' + gradeRes + ' to autogradeResult.txt'))
428             appendFileDat('autogradeResult.txt', gradeRes)
429
430
431     #this will process input configuration and return a dictionary holding the relevant
        info
432     def buildBFConfigStruct(args):
433         import time
434         bfConfigData = readFileDat(args.configFileName)
435         configData = dict()
436         for line in bfConfigData:
437             #build dictionary on non-list elements
438             if (line[0]=='#') or ('_' in line):
439                 continue
440             elems = line.split('=')
441             if('name' in elems[0]):
442                 configData[elems[0]]=elems[1].strip()
443             else :
444                 configData[elems[0]]=int(elems[1])
445
446         if ('Type 1' in configData['name']):
447             configData['type'] = 1
448             configData['seeds'] = buildSeedList(bfConfigData, int(configData['k']))
449
450         elif ('Type 2' in configData['name']):
451             configData['type'] = 2
452             aListData = []
453             bListData = []
454             listToAppend = aListData
455             for line in bfConfigData:
456                 if (line[0]=='#'):
457                     if ('b() seeds' in line):
458                         listToAppend = bListData
459                     continue
460                 listToAppend.append(line)
461
462             configData['a']= buildSeedList(aListData, int(configData['k']))
463             configData['b']= buildSeedList(bListData, int(configData['k']))
464         else :
465             configData['type'] = -1
466             print('unknown hash function specified in config file')
467
468         configData['task'] = int(args.taskToDo)
469         if configData['task'] != 2 :
470             configData['genSeed'] =  int(time.time()*1000.0) & 0x7FFFFFFF
                #(int)(tOffLong & 0x7FFFFFFF);
471             print(('Random Time Seed is : ' + str(configData['genSeed'])))
472
473         configData['inFileName'] = args.inFileName
474         configData['outFileName'] = args.outFileName
475         configData['configFileName'] = args.configFileName
476         configData['valFileName'] = args.valFileName
477         configData['studentName'] = args.studentName
478         configData['autograde'] = int(args.autograde)
479
480         for k,v in list(configData.items()):
481             print(('Key = ' + k + ': Val = '), end=' ')
482             print(v)
483
484         return configData
485
486     def buildSeedList(stringList, k):
487         res = [0 for x in range(k)]
488         for line in stringList:
489             if ('_' not in line) or (line[0]=='#'):
490                 continue
```

```
491             elems = line.split('=')
492             araElems = elems[0].split('_')
493             res[int(araElems[1])]=int(elems[1])
494         return res
495 
496 
497     """
498         Function provided for convenience, to find next prime value from passed value
499         Use this to find an appropriate prime size for type 2 hashes.
500 
501         Finds next prime value larger than n via brute force.  Checks subsequent numbers
502         until prime is found - should be much less than 160 checks for any values
503         seen in this project since largest gap g between two primes for any 32 bit
504         signed int is going to be g < 336, and only have to check at most every
505         other value in gap. For more, see this article :
506             https://en.wikipedia.org/wiki/Prime_gap
507 
508         n  : some value
509         return next largest prime
510     """
511     def findNextPrime(n):
512         if (n==2) :
513             return 2
514         if (n%2==0):
515             n+=1
516         #n is odd here; 336 is larger than largest gap between 2 consequtive 32 bit primes
517         for i in range (n,(n + 336), 2):
518             if checkIfPrime(i):
519                 return i
520         #error no prime found returns -1
521         return -1
522 
523     """
524         check if value is prime, return true/false
525         n value to check
526     """
527     def checkIfPrime(n):
528         if (n < 2) : return False
529         if (n < 4) : return True
530         if ((n % 2 == 0) or (n % 3 == 0)): return False
531         sqrtN = n**(.5)
532         i = 5
533         w = 2
534         while (i <= sqrtN):
535             if (n % i == 0): return False
536             i += w
537             #addresses mod2 and mod3 above, flip flops between looking ahead 2 and 4
                (every other odd is divisible by 3)
538             w = 6-w
539         return True
540 
541     ## end bloom filter functions
542     ####################################
543 
544     ########################################
545     #Page Rank Functions
546 
547 
548     #get file values for particular object and alpha value
549     #results are list of nodes, list of rank values and dictionary matching node to rank
    value
550     #list of nodes and list of rank values are sorted
551     def getResForPlots(prObj, alpha):
552         outFileName = makeResOutFileName(prObj.inFileName, alpha, prObj.sinkHandling)
553         vNodeIDs_unsr, vRankVec_unsr = loadRankVectorData(outFileName, isTest=False)
554         #build dictionary that links node id to rank value
555         vNodeDict = buildValidationDict(vNodeIDs_unsr,vRankVec_unsr)
556 
557         #build sorted list
558         vNodeIDs, vRankVec = getSortResIDXs(vRankVec_unsr)
559         return vNodeIDs, vRankVec, vNodeDict
560 
561     #build appropriate results file name based on passed input name, alpha and sink
```

```
    handling flag
562 def makeResOutFileName(inFileName,alpha,sinkHandling):
563     nameList = inFileName.strip().split('.')
564     namePrefix = '.'.join(nameList[:-1])
565     #build base output file name based on input file name and whether or not using
        selfloops to handle sinks
566     outFileName = "{}_{}_{}.{}".format(namePrefix,("SL" if sinkHandling==0 else
        "T3"), alpha,nameList[-1])
567     return outFileName
568
569 #builds output file names given passed file name
570 def buildPROutFNames(fName, getVerifyNames=False):
571     #construct ouput file names based on fName (which is input file name : i.e.
        'inputstuff.txt')
572     nameList = fName.strip().split('.')
573     #name without extension
574     namePrefix = '.'.join(nameList[:-1])
575     if getVerifyNames :
576         #get names for verification files
577         #file holding rank vector values
578         voutFName = '{}-{}.{}'.format(namePrefix, 'verifyRVec',nameList[-1])
579         return voutFName
580     else :
581         #names for saving results or accessing saved results
582         #file holding rank vector values
583         outFName = '{}-{}.{}'.format(namePrefix, 'outputPR',nameList[-1])
584         return outFName
585
586
587 #this will build a dictionary with :
588 #   keys == graph nodes and
589 #   values == list of pages accessible from key
590 #   and will also return a list of all node ids
591 #   using terminology from lecture, this builds the "out list" for each node in
592 #   file, and a list of all node ids
593 def loadGraphADJList(fName):
594     from collections import defaultdict
595     #defaultDict has 0/empty list entry for non-present keys,
596     #does not return invalid key error
597     resDict = defaultdict(list)
598     filedat = readFileDat(fName)
599     allNodesSet = set()
600     #each line has a single number, followed by a colon, followed by a list of
601     #1 or more numbers spearated by commas
602     #these represent node x : reachable nodes from node x
603     for line in filedat:
604         vals = line.strip().split(':')
605         adjValStrs = vals[1].strip().split(',')
606         #convert list of strings to list of ints
607         adjVals = [int(s.strip()) for s in adjValStrs]
608         key = int(vals[0].strip())
609         allNodesSet.add(key)
610         allNodesSet.update(adjVals)
611         resDict[key] = adjVals
612     return resDict, list(allNodesSet)
613
614 #given the base input file name
615 #this will return a list of nodes in order of rank (if rankName file exists)
616 #and a vector of rank values as floats (if outputName file exists)
617 #using either base file extensions or the verification file names
618 def loadRankVectorData(fName, isTest=False):
619     outFName = buildPROutFNames(fName, isTest)
620     #read rank vector as list of floats, expected to be in order of node ids
621     rankVec = readFloatFileDat(outFName)
622
623     rankedIDS = list(range(len(rankVec)))
624     #either output, or both, might be empty list(s) if files don't exist
625     return rankedIDS, rankVec
626
627
628 #will save a list of nodes in order of rank, and rank values (the rank vector) for
    those nodes in same order
629 #in two separate files
```

```
630 def saveRankData(fName, rankVec=None):
631     outFName = buildPROutFNames(fName)
632 
633     if(rankVec != None):
634         writeFileDat(outFName, rankVec)
635         print(('Rank vector saved to file {}'.format(outFName)))
636 
637 
638 #build a dictionary that will have node id as key and rank vector value as value - used for verification since equal rank vector values might be in different order
639 def buildValidationDict(nodeIDs, rankVec):
640     vDict = {}
641     for x in range(len(nodeIDs)):
642         vDict[nodeIDs[x]] = rankVec[x]
643     return vDict
644 
645 """
646 using provided output file, verify calculated page rank is the same as expected results
647 args used for autograder version
648 """
649 def verifyResults(prObj, args=None, eps=.00001):
650     print(('\nVerifying results for input file "{}" using alpha={} and {} sink handling :\n'.format(prObj.inFileName, prObj.alpha, ('self loop' if prObj.sinkHandling==0 else 'type 3'))))
651     #load derived values from run of page rank
652     calcNodeIDs,calcRankVec = loadRankVectorData(prObj.outFileName, isTest=False)
653     #load verification data
654     vNodeIDs, vRankVec = loadRankVectorData(prObj.outFileName, isTest=True)
655     if (len(vNodeIDs) == 0) or (len(vRankVec)==0) :
656         print ('Validation data not found, cannot test results')
657         return False
658 
659 
660     #compare nodeID order
661     if(len(calcNodeIDs) != len(vNodeIDs)) :
662         print(('!!!! Error : incorrect # of nodes in calculated page rank - yours has {}; validation has {}'.format(len(calcNodeIDs),len(vNodeIDs))))
663         return False
664     print('Calculated Rank vector is of appropriate length')
665 
666     #need to verify that rank vector sums to 1
667     cRVecSum = sum(calcRankVec)
668     if abs(cRVecSum - 1) > eps :
669         print(('!!!! Error : your calculated rank vector values do not sum to 1.0 : {} '.format(cRVecSum)))
670         return False
671     print('Calculated Rank vector has appropriate magnitude of 1.0')
672 
673     #build dictionary of validation data and test data - doing this because order might be different for nodes with same rank value
674     validDict = buildValidationDict(vNodeIDs,vRankVec)
675     calcDict = buildValidationDict(calcNodeIDs,calcRankVec)
676 
677 
678     #compare if matched - Note nodes with same rank value vector value might be out of order
679     for x in range(len(vNodeIDs)):
680         if abs(calcDict[vNodeIDs[x]] - validDict[vNodeIDs[x]]) > eps :
681             print(('!!!! Error : rank vector values do not match, starting at idx {}, node {}, in validation node id list'.format(x,vNodeIDs[x])))
682             return False
683     print('Rank Vector values match verification vector values')
684 
685     return True
686 
687 #autograder code
688 def autogradePR(prObj, args, prMadeTime):
689     print(('Running autograder on {} for prObj with input file {}'.format(args.studentName, prObj.inFileName)))
690 
691 
692 #End Page Rank Functions
```

```
693 ##########################################################################
694 # Start findXinA Functions  (Added Summer 2020 rockograziano@gatech.edu
695
696 import random
697 import math
698 import sys
699
700 class ExceededLookupsError(Exception):
701     def __init__(self, *args):
702         if args:
703             self.message = args[0]
704         else:
705             self.message = None
706
707     def __str__(self):
708         if self.message:
709             return '{0}'.format(self.message)
710         else:
711             return 'ExceededLookups: Program Exceeded the allowed number of lookups'
712
713
714 class findX():
715     def __init__(self):
716         self.__A = []
717         self.__n = 0
718         self.x = 0
719         self.__lookupCount=0
720         self.__maxCalls = 0
721         return
722
723     def start(self, seed, nLower=10, nUpper=100000):
724         random.seed(seed)
725         self.__lookupCount=0
726         self.__n = random.randint(nLower, nUpper)
727         self.__A = random.choices(range(-nUpper*2,nUpper*2), k=self.__n+1) # sample
            extra value to avoid A[n] error
728         self.__A.sort()
729         self.x = self.__A[random.randint(1,self.__n)]
730         self.__maxCalls = int(math.log(self.__n, 2)*2) + 2
731         return self.x
732
733     def lookup(self, i):
734         if not isinstance(i, int):
735             raise TypeError('x must be an integer')
736
737         if i < 1:
738             raise ValueError('x must be > 0')
739
740         self.__lookupCount += 1
741
742         if self.__lookupCount > self.__maxCalls:
743             raise ExceededLookupsError('Exceeded Maximum of {}
                Lookups'.format(self.__maxCalls))
744
745         if i > self.__n:
746             return None
747         else:
748             return self.__A[i]
749
750     def lookups(self):
751         return self.__lookupCount
752
753 #End findXinA functions
754 ################################
755
756
```