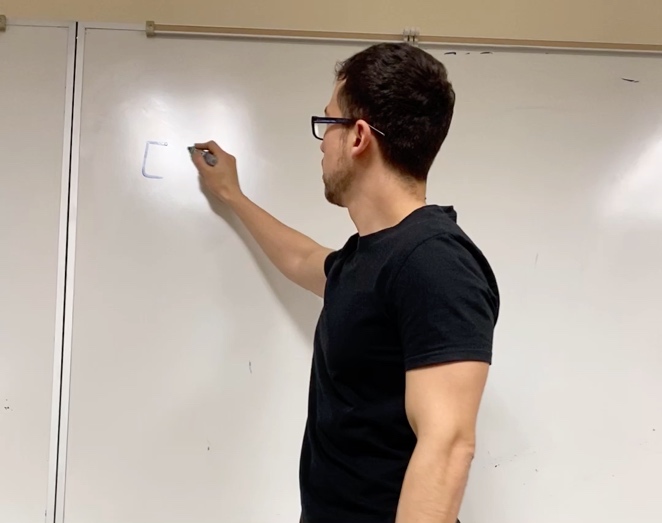
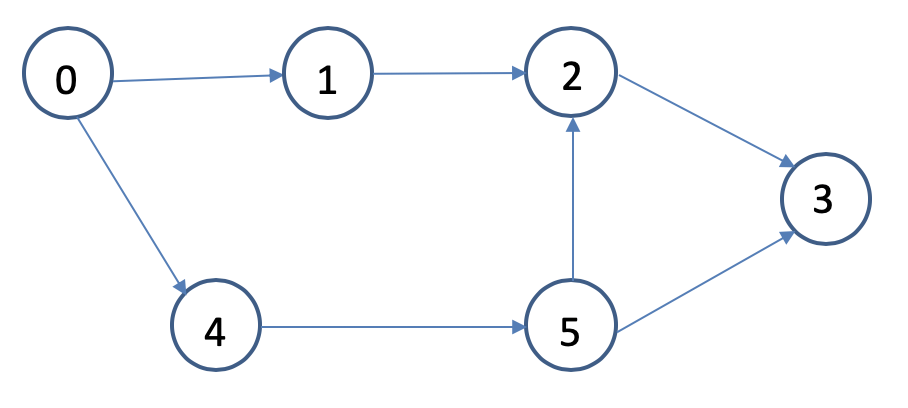
**Data Structures - Fall 2019**

**Practice Exam 3**

**Section 1 - Tracing**

s1\_tracing.py

**Problem 1 (5 points):** Trace the execution of the topological sort algorithm using the graph below as input. Break ties by always enqueuing the vertex with the smallest index first. The result of topological sort is an array. After tracing, return the contents of this array in get\_problem\_1\_answer()

****

**def** topological\_sort(graph):

all\_in\_degrees = compute\_indegree\_every\_vertex(graph)

sort\_result = []

q = Queue()

**for** i **in** range(len(all\_in\_degrees)):

**if** all\_in\_degrees[i] == 0:

q.put(i) *# enqueue*

**while not** q.is\_empty():

u = q.put() *# dequeue*

sort\_result.append(u)

**for** adj\_vertex **in** graph.get\_adj\_vertices(u):

all\_in\_degrees[adj\_vertex] -= 1

**if** all\_in\_degrees[adj\_vertex] == 0:

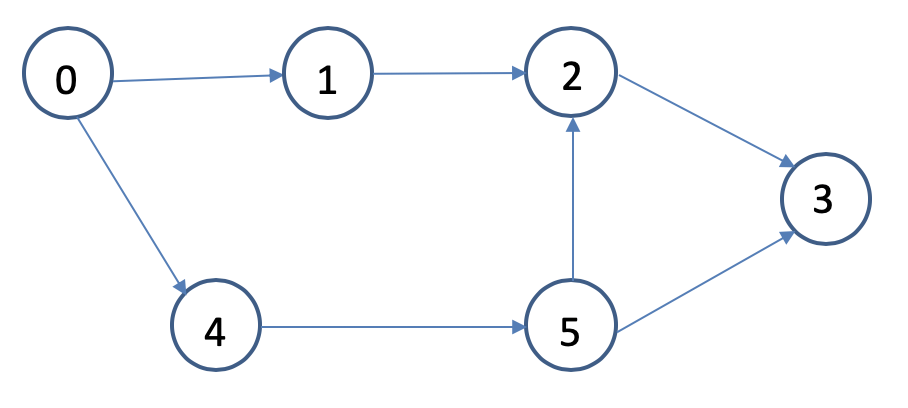
q.put(adj\_vertex)

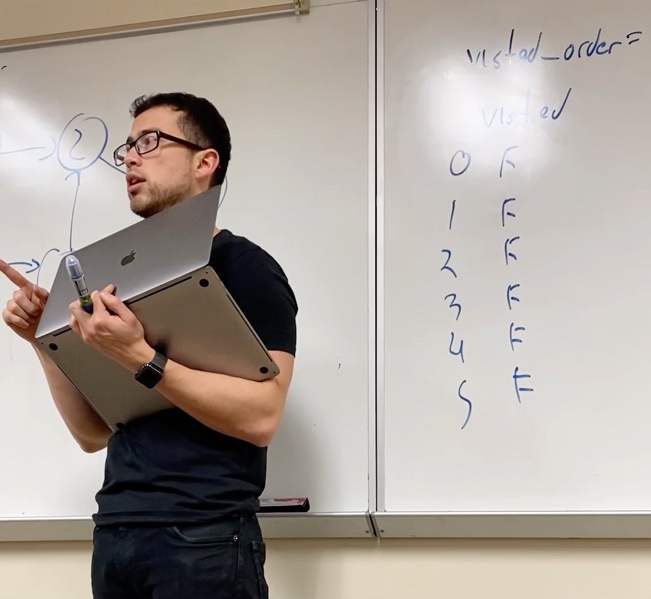
**if** len(sort\_result) != graph.num\_vertices: *# Cycle found*

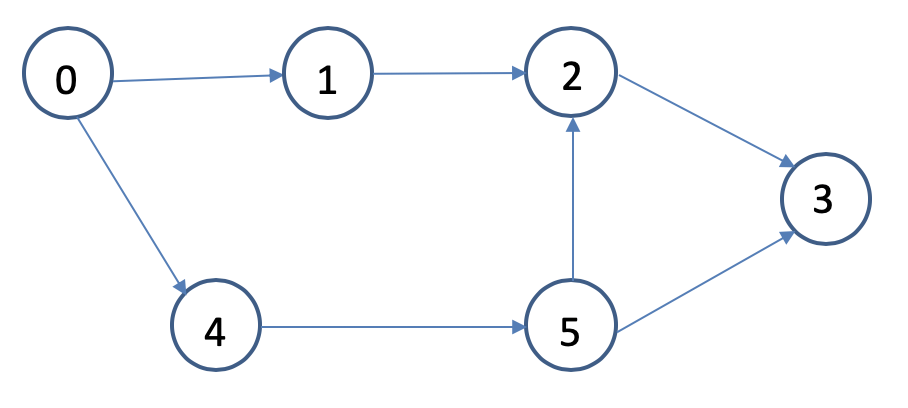
**return None**

**return** sort\_result

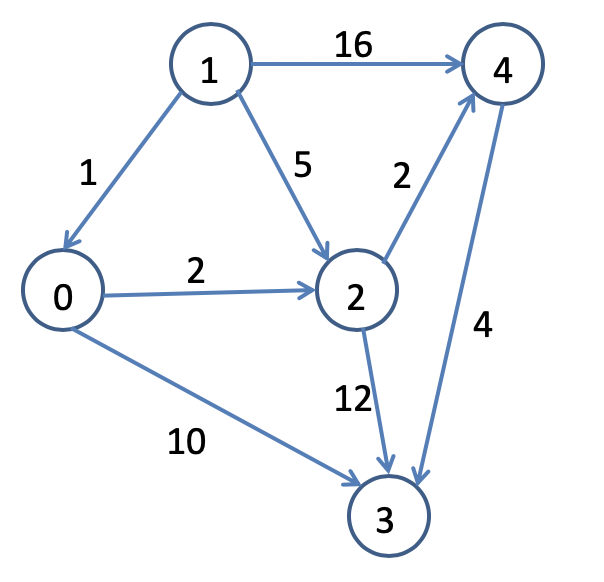
**Problem 2 (5 points):** Trace the execution of breadth-first search using the graph below as input. Start from vertex 0 and break ties by always pushing/enqueuing the vertex with the smallest index first. After tracing, return the contents of the visited\_array (the array that stores the order in which the vertices are visited) in get\_problem\_2\_answer()



**Problem 3 (5 points):** Trace the execution of depth-first search using the graph below as input. Start from vertex 0 and break ties by always pushing/enqueuing the vertex with the smallest index first. After tracing, return the contents of the visited\_array (the array that stores the order in which the vertices are visited) in get\_problem\_3\_answer()



**Problem 4 (8 points):** Trace the execution of Dijkstra's algorithm to determine the shortest path from vertex 1 to every other vertex in the graph. After tracing, return the contents of the *dist* and *path* arrays in get\_problem\_4\_answer()



**def** dikstra(graph, src):

known = [**False**] \* graph.num\_vertices

path = [-1] \* graph.num\_vertices

dist = [math.inf] \* graph.num\_vertices

dist[src] = 0

known\_vertices = 0

**while** known\_vertices < graph.num\_vertices:

u = get\_unknown\_vertex\_smallest\_dist(graph, known, dist)

known[u] = **True**

known\_vertices += 1

**for** v **in** get\_unknown\_neighbors(u, graph, known): *# unknown neighbors of u*

**if** dist[v] > dist[u] + graph.edge\_weight(u, v):

path[v] = u

dist[v] = dist[u] + graph.edge\_weight(u, v)

**return** path, dist

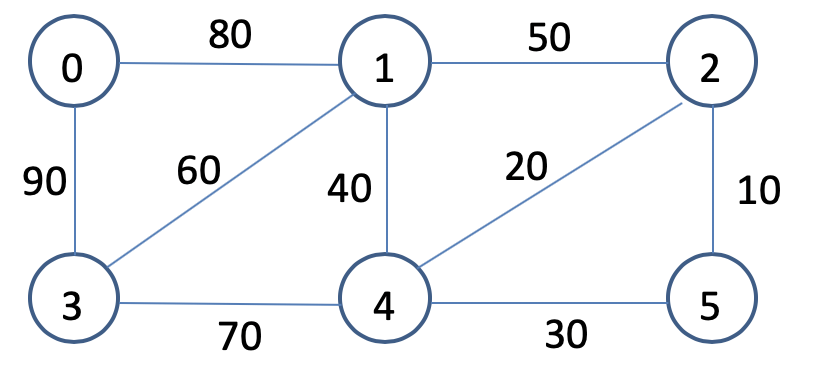
**Problem 5 (5 points):** Apply the edit distance algorithm based on dynamic programming to determine the minimum sequence of elementary character operations needed to convert the string "STACK" to the string "SMART". Return the contents of the 2D array in get\_problem\_5\_answer()

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | ‘’ | S | T | A | C | K |
| ‘’ |  |  |  |  |  |  |
| S |  |  |  |  |  |  |
| M |  |  |  |  |  |  |
| A |  |  |  |  |  |  |
| R |  |  |  |  |  |  |
| T |  |  |  |  |  |  |

**Section 2 - Multiple Choice**

s2\_multiple\_choice.py

**Problem 6 (5 points):** Consider the following graph:



Which of the following edges should **not** be part of the graph’s *minimum spanning tree?*

Option 0: Edge connecting 3 and 4

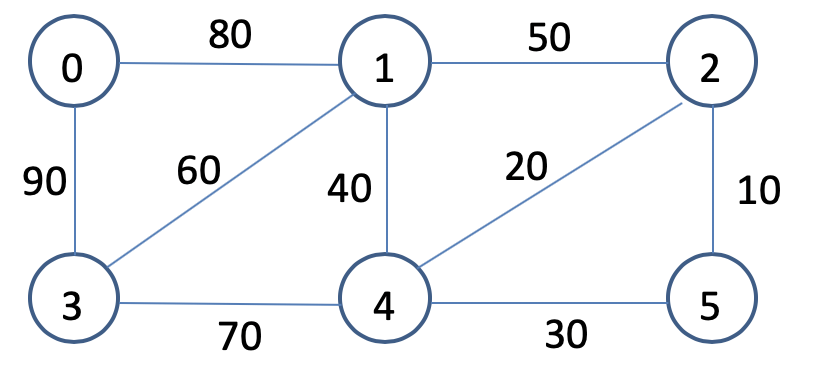
Option 1: Edge connecting 1 and 4

Option 2: Edge connecting 0 and 1

Option 3: Edge connecting 3 and 1

Option 4: None of the above

**Problem 7 (5 points):** Consider the following graph (same as before):



How many edges are there in the graph’s minimum spanning tree?

Option 0: 6

Option 1: 5

Option 2: 7

Option 3: 4

Option 4: None of the above

**Problem 8 (3 points):** What data structure does breadth-first search use to traverse a graph:

Option 0: Min-Heap

Option 1: Queue

Option 2: Stack

Option 3: Disjoint Set Forest

Option 4: None of the above

**Problem 9 (3 points):** What data structure does depth-first search use to traverse a graph:

Option 0: Min-Heap

Option 1: Queue

Option 2: Stack

Option 3: Disjoint Set Forest

Option 4: None of the above

**Problem 10 (3 points):** Which of the following algorithms uses divide and conquer?

Option 0: Binary Search

Option 1: Quicksort

Option 2: Merge Sort

Option 3: All of the above

Option 4: None of the above

**Problem 11 (3 points):** How does the randomized version of quick sort work?

Option 0: It calls itself recursively at random locations in the array

Option 1: It uses randomly generated numbers to traverse the unsorted array

Option 2: It randomly selects a pivot from the array

Option 3: It randomly swaps elements in the array before sorting it

Option 4: None of the above

**Problem 12 (3 points):** Which of the following algorithm(s) use dynamic programming

Option 0: Kruskal’s algorithm

Option 1: Prims’s algorithm

Option 2: Dijkstra’s algorithm

Option 3: Edit Distance

Option 4: Bubble sort

Option 5: Options 0 and 1

Option 6: Options 2 and 3

Option 7: None of the above

**Problem 13 (3 points):** (Fill in the blank) Greedy algorithms \_\_\_\_\_\_\_ find the optimal solution to the problem they attempt to solve

Option 0: Always

Option 1: Never

Option 2: Sometimes

Option 3: None of the above

**Problem 14 (5 points):** Which of the following statements is true?

Option 0: All NP-Complete problems are also NP problems

Option 1: All NP problems are also NP-Complete problems

Option 2: All NP-Hard problems are also NP problems

Option 3: All NP problems are also NP-Hard

Option 4: None of the above

**Section 3 - Graphs - Adjacency List Representation**

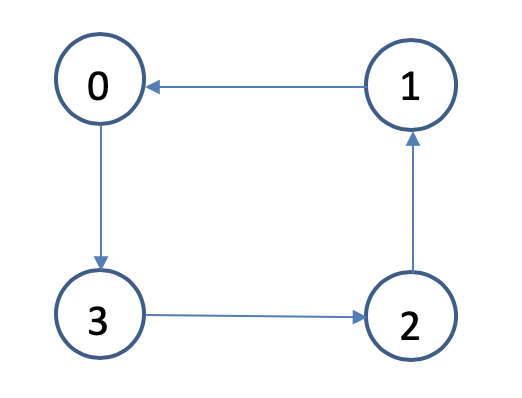
s3\_graph\_al.py

**Problem 15 (5 points):** Complete the implementation of the *is\_there\_an\_edge* method. This method should return True if there is an edge going from *src* to *dest;* False, otherwise.

**Problem 16 (5 points):** Complete the implementation of the *compute\_out\_degree* method. The *out degree* of a vertex *v* in a directed graph *G = (V,E)* is the number edges coming out of *v*. The method *compute\_out\_degree* should receive a vertex *v* and and return the *out degree* of *v*.

**Problem 17 (6 points):** Complete the implementation of the *is\_isolated* method. An isolated vertex is a vertex with no ingoing or outgoing edges. The method *is\_isolated* should receive a vertex *v* and return True if *v* isolated vertex in the graph; False, otherwise.

**Problem 18 (8 points):** A backward circle graph of size n is a graph that contains n vertices and (only) the edges (1,0),(2,1),..., (n-1,n-2),(0, n-1). The figure shows an example of a circle graph of size 4. Complete the implementation of *create\_backward\_circle\_graph* method.



**Section 4 - Graphs - Adjacency Matrix Representation**

s4\_graph\_am.py

**Problem 19 (5 points):** Complete the implementation of the *is\_there\_an\_edge* method. This method should return True if there is an edge going from *src* to *dest;* False, otherwise.

def p19(self, src, dest): \*\*\*Adjacency List

if ------

return False

lst = self.al[src]

for edge in lst:

if edge.dest==dest:

return True

return False

def p19(self, src, dest): \*\*\*Adjacency Matrix

if ------

return False

return self.am[src][dest] != 0

**Problem 20 (5 points):** Complete the implementation of the *compute\_out\_degree* method. The *out degree* of a vertex *v* in a directed graph *G = (V,E)* is the number edges coming out of *v*. The method *compute\_out\_degree* should receive a vertex *v* and and return the *out degree* of *v*.

**Problem 21 (6 points):** Complete the implementation of the *is\_isolated* method. An isolated vertex is a vertex with no ingoing or outgoing edges. The method *is\_isolated* should receive a vertex *v* and return True if *v* isolated vertex in the graph; False, otherwise.

def is\_isolated(self, v): \*\*Adjacency List

if ---- valid

return false

if len(self.al[v]) != 0:

return False

for lst in self.al:

for edge in lst:

if edge.dest == v

return False

return True

A close up of text on a whiteboard

Description automatically generated

def is\_isolated(self, v): \*\*Adjacency Matrix

if ---- valid

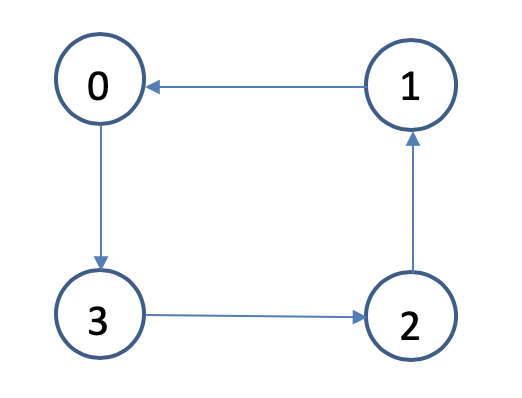
return false

for i in range (len(self.am)):

if self.am[i][v] != 0 or self.am[v][i] != 0:

return false.

**Problem 22 (8 points):** A backward circle graph of size n is a graph that contains n vertices and (only) the edges (1,0),(2,1),..., (n-1,n-2),(0, n-1). The figure shows an example of a circle graph of size 4. Complete the implementation of *create\_backward\_circle\_graph* method.



def p22(n):

Graph = GraphAL(directed = True, weighted = False, vertices = n):

for i in range of (1, n):

graph.insert\_edge(i, i-1):

graph.insert\_edge(0,n-1)

return graph

**Section 5 – Disjoint Set Forest**

s5\_dsf.py

**Problem 23 (5 points):** Complete the implementation of the *get\_num\_sets* method. This method is to return the number of sets in the disjoint set forest

**Problem 24 (5 points):** Complete the implementation of the *create\_dsf* method. This method receives an integer n and builds and returns an array (that represents a disjoint set forest) of size n. This disjoint set forest should encode n / 2 sets, where every pair of subsequent numbers belong to the same set. The bigger element must point to the smaller one.

Example: n = 8, {-1, 0, -1, 2, -1, 4, -1, 6}

dsf p24(n):

dsf = -1 \* n

for i in range (1,n,2):

dsf[i] = i-1

return dsf

///// sets # = n/3  **// both**

for i in range (1,n):

if i%3! = 0:

dsf[i] = i – 1 **(dsf[i] = (i//3)\*3) \*for** **compressed set**

return dsf