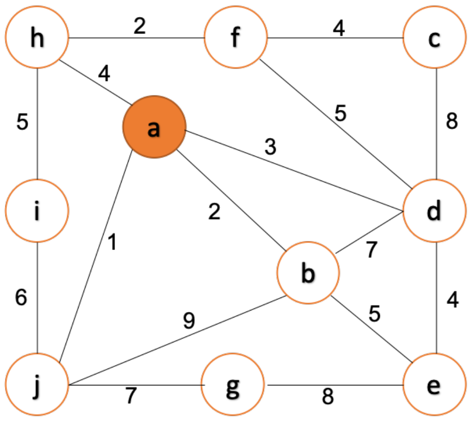
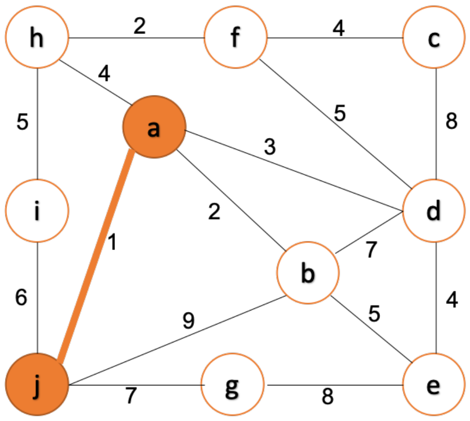
Problem #1

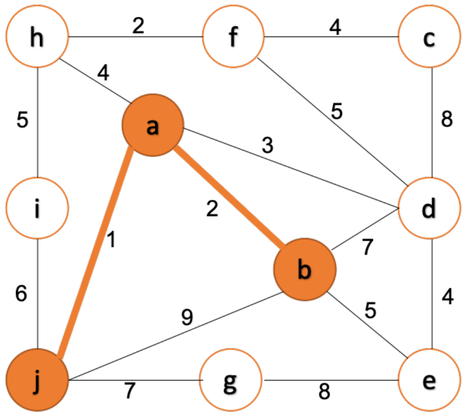
(1)



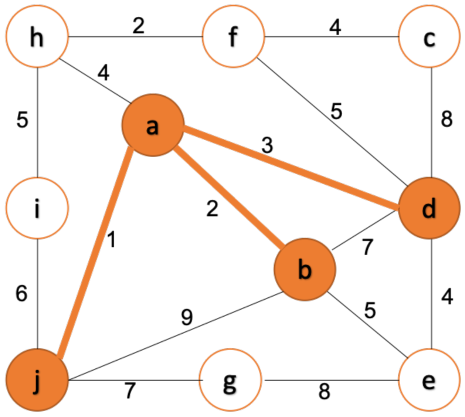
(2)



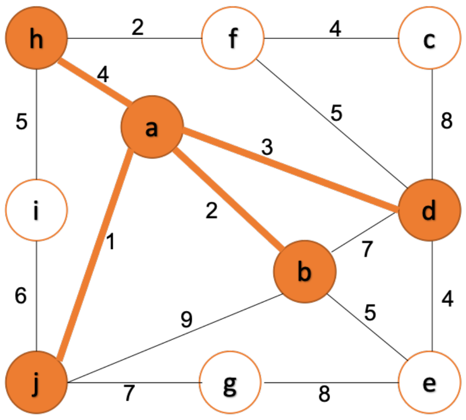
(3)



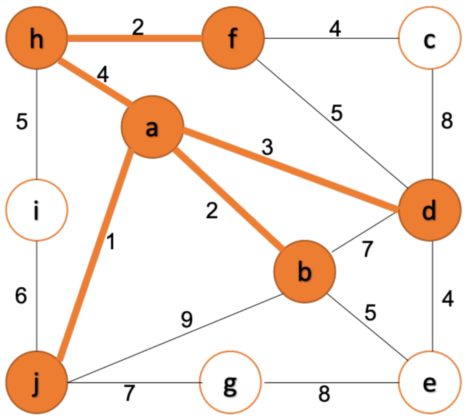
(4)



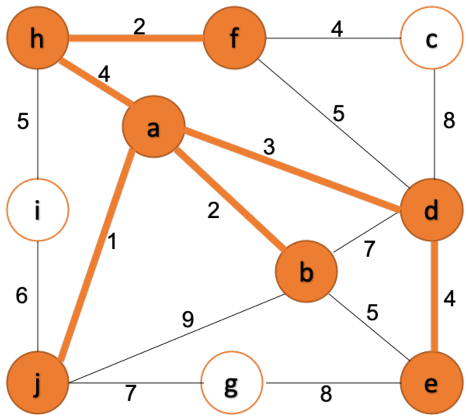
(5)



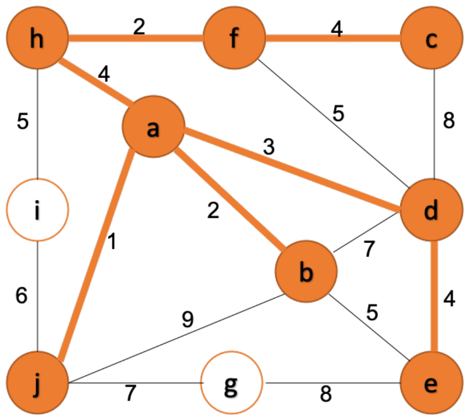
(6)



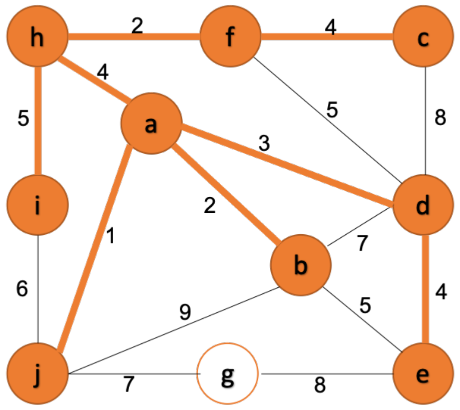
(7)



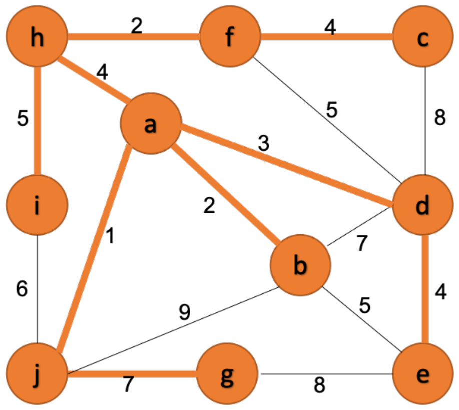
(8)



(9)



(10)



Weight of the minimum spanning tree: 1+2+2+3+4+4+4+5+7 = 32

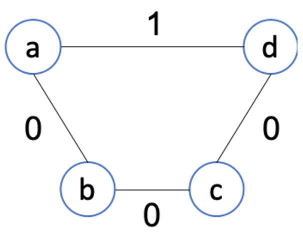
Problem #2

(a)
We have computed one minimum spanning tree of G. Let's say the mst is T. T has a unique set of weight of edges, [e1, e2, e3,…., e(n-1)] (because we have only one mst) in increasing order where G has n vertices.

Let's say we used Kruskal's algorithm to compute the minimum spanning tree. Assume we add 1 to each edge of the MST, then we have [e1+1, e2+1, e3+1,…., e(n-1)+1] which preserves its order, and we obviously get the same minimum spanning tree using the algorithm.

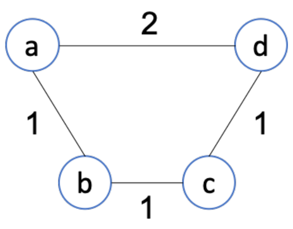
Therefore, we will have the same minimum spanning tree. (No change)

(b)
Yes, the shortest paths could change.



The shortest path is [a,b,c,d] with total weight of 0.

If we add 1 to each edge, the shortest path changes to [a,d] with total weight of 2.



Problem #3

(a)
We can solve this problem by modifying Prim’s or Kruskal’s algorithm.

When using Prim’s, we can simply skip the minimum value extracted from Q if the value is less than W. So, let’s say W is 4. If ExtractMin(Q) returns 3, then we do nothing but extract the next minimum value in Q.

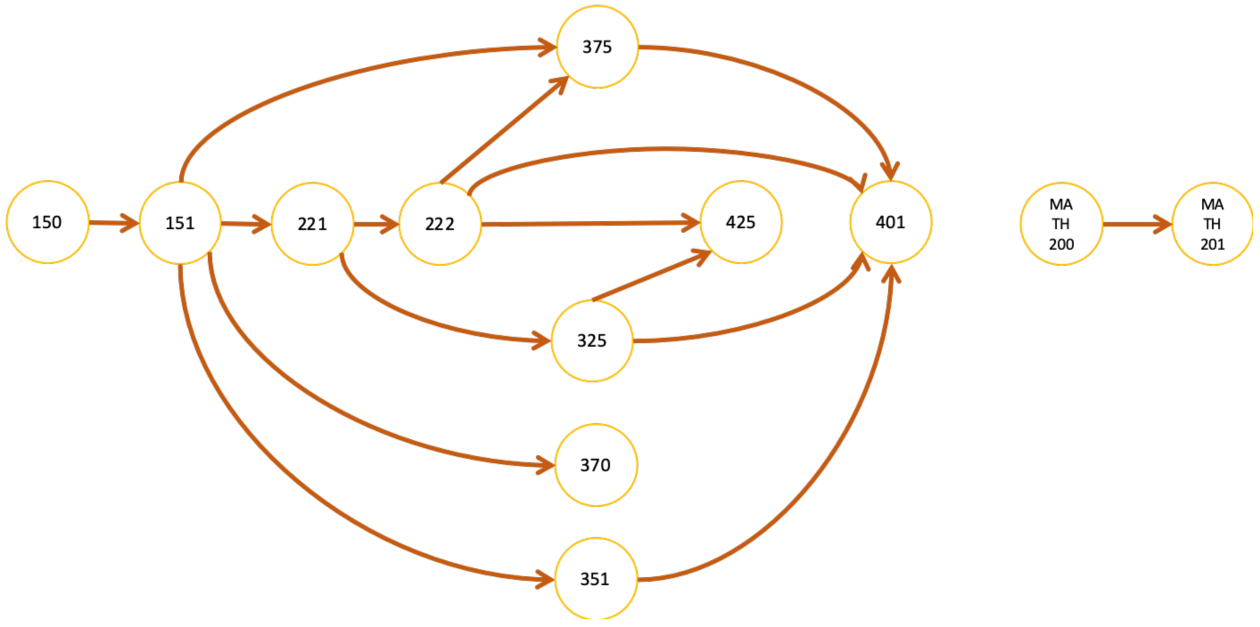
When using Kruskal’s, after sorting E by increasing edge weight w, we can put a if statement in the loop checking every edge, which checks the weight of edge is less than W. If it isn’t, we ignore the edge.

For the last step, we should count the total number of edge if it equals to n-1 where n is the number of vertices of the graph. If it’s not, there’s no path found.

(b)
If using a binary heap for container, the running time is O(ElogV) where E is the number of edges and V is the number of vertices.

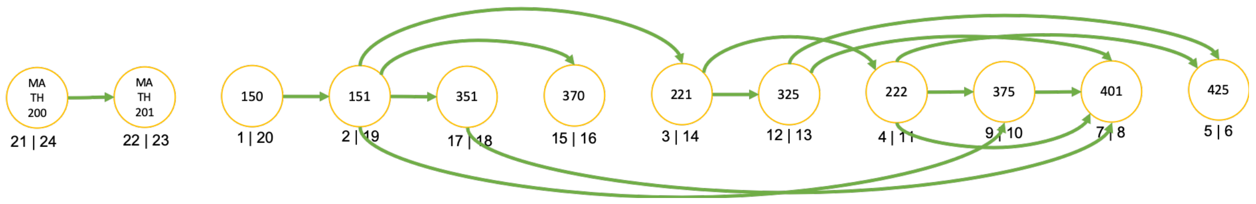
Problem #4

(a)



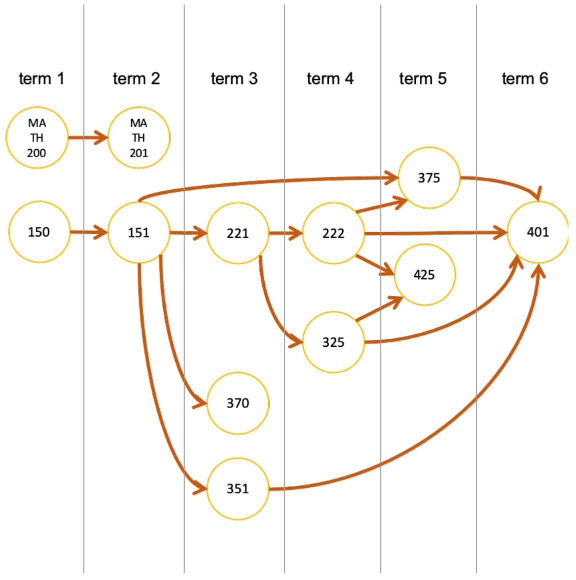
(b)

In order of decreasing finishing time.



(c)

A student can take all classes in 6 terms.



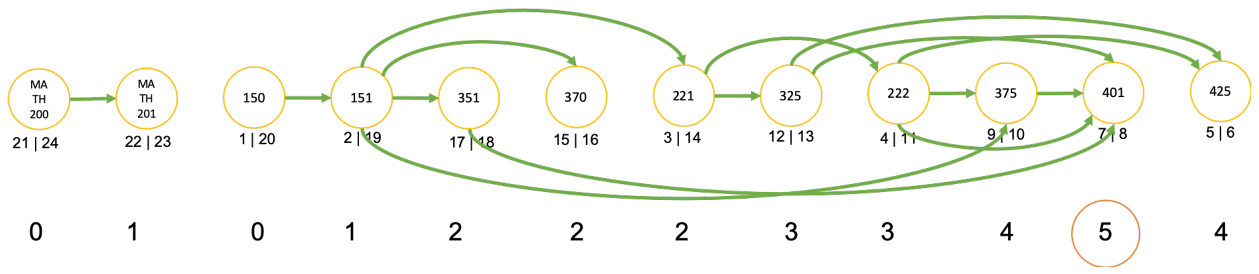
(d)

Longest path of the graph: 150 – 151 – 221 – 222 – 375 – 401 So, the length of the longest path in the DAG is 5.

To get the longest path in the DAG,

1) Topologically sort the graph
2) Initialize the distances associated with vertices in the graph to 0. That is d(v)=0 for all v in G.
3) Start with the first vertex v in the topological sort.
   For each u in Adj[v] set d(u) = max { d(u), d(v)+1 }
4) Repeat step 4 with the next vertex in the topological sorted order until all vertices have been examined.

According to this algorithm, we have 5, the length of the longest path.



The length of the path + 1, 5 + 1 = 6, represents the fewest number of terms required to complete all courses with no prerequisite conflicts.

Problem #5
(a)

n = the number of vertices
graph = an adjacency n x n matrix that represent the graph. If there's an edge between two vertices, cell is filled with 1, otherwise 0.
marks = [1,2, …, n] *// Initialize an array with -1, which stores status of each vertex in integer, -1: unchecked, 0: Blue, 1: Red*
bipartite = True *// boolean value*

```
// Perform BFS search to mark a color to vertices
isBipartite(n, marks, graph, i)
    queue = []              // a queue for BFS search
    queue.append(i)    // Store a value to queue

    while queue is not empty
        u = queue.pop()         // dequeue the queue

        for v from 1 to n

            // if there's an edge between u and v and v is not checked
            if graph[u][v] == 1 && marks[v] == -1
                marks[v] = 1 – marks[u]        // Assign different color to v than u
                queue.append(v)

            // if there's an edge between u and v and two vertices have the same color
            else if graph[u][v] == 1 and marks[v] == marks[u]:
                return False                       // Can't designate the valid rivalries

      return True                                  // Can designate the rivalries

// Main program
for i from 1 to n
   if marks[i] == -1    // if vertex is unchecked
      marks[i] = 0   // assign it with Blue
      bipartite = isBipartite(n, marks, graph, i)

    // Print
   if bipartite
      babyfaces = [list of wrestlers marked 0, Blue]
      heels = [list of wrestlers marked 1, Red]
      print babyfaces and heels
  else
       print No
```

(b)
O(n) where n is the number of wrestlers. Once a wrestler is assigned to a team, we only need to check the unassigned wrestlers, and no need to look back the assigned one. So, it takes O(n) times.

<citations>
https://en.wikipedia.org/wiki/Minimum_spanning_tree
https://en.wikipedia.org/wiki/Widest_path_problem
graphPracticesol.pdf (class material)
https://www.programiz.com/dsa/graph-adjacency-matrix
https://www.geeksforgeeks.org/bipartite-graph/