Name: Jungbin Choi COMP 550-001
Due September 22, 2023 Homework 2 (14 points)

Notes. Collaboration is allowed, and you may use any resources you'd like. You may call any algorithm given in the notes without providing details. However, you should write and submit your solutions individually, and each submission will be graded individually. In your submission, you should list your collaborators (including course staff) and cite every resource you use. For each problem, I encourage you to attempt it for at least 30 minutes before consulting any people or resources.

Your submission should be a single PDF file. I recommend using Overleaf; a template is on Canvas. If your answers are not clear, you might not get full credit. In general, you should describe your algorithms in pseudocode similar to the examples given in the lectures/notes. I recommend also including a brief description in full sentences.

You may call any algorithms given in the notes or lecture without providing the details.

Problem 1 (3 points). Let $G = (V, E)$ be a connected, undirected graph containing n vertices and m edges. For any $u, v \in V$, let $d(u, v)$ denote the distance between u and v. (Recall that the distance between u and v is the length of the shortest path from u to v.) The *diameter* of G is the maximum value of $d(u, v)$ taken over all $u, v \in V$.

a. (2p) Give an $O(mn)$-time algorithm that returns the diameter of G.

b. (1p) Analyze the running time of your algorithm.

Solution.

a.
```
ALG(G):
    diameter = 0
    max[] = [-1] * n
    for v in G:
        dist[] = [-1] * n
        dist[v] = 0
        Q = new Queue()
        Q.add(v)
        while Q is not empty:
            current = Q.dequeue()
            for each neighbor of current:
                if dist[neighbor] == -1:
                    dist[neighbor] = dist[current] + 1
                    Q.add(neighbor)
        max_d = 0
        for d in dist:
            if d > max_d:
                max_d = d
        max[v] = max_d
    for d in max:
        if d > diameter:
            diameter = d
    return diameter
```

b. All assignments, arithmetic, indexing, and comparisons cost $O(1)$ time, therefore will not be mentioned from here on. $max[]$ initialization costs $O(n)$. The 1st for-loop iterates through every vertex, thus $O(n)$ iterations. Per iteration, $dist[]$ initialization costs $O(n)$. The while-loop is a bit tricky: inside the loop will cost $O(\#neighbors)$ of each vertex visited, and when summed up it'll lead to $O(2m)$ because each edge will be counted twice; therefore, the while-loop itself is $O(2m)$. Next, the 2nd for-loop will cost $O(n)$ because it'll iterate through $dist[]$ once. In total, the 1st for-loop will cost $O(n(n + 2m + n))$, which will simplify down to $O(n^2 + mn)$. Then, the last for-loop will cost $O(n)$ because it iterates through $max[]$ once. Finally, the total running time will be the sum of the initialization of $max[]$, the 1st for-loop, and the last for-loop. This total is $O(n+(n^2+mn)+n)$ or simplified to $O(mn)$ as in the worst case scenario, m = every vertex connected to every other vertex = $(n(n-1))/2$.

Problem 2 (4 points). Suppose there is a cooking competition with n dishes and k judges. Each judge ranks exactly ℓ of the n dishes from tastiest to least tasty (in their opinion). Our goal is to rank all n dishes in a way that is consistent with every judge's ranking. If this task is not possible (due to disagreement among the judges), our algorithm should return 0.

More formally, the input is n and an array A of size k, where $A[i]$ is a list containing the ℓ dishes ranked by judge i from tastiest to least tasty. Each dish is represented by a number in $\{1, \ldots, n\}$. The output should either be a list of the n dishes consistent with all judges' rankings, or the integer 0.

a. (2p) Give an $O(k\ell + n)$-time algorithm for this problem.

b. (1p) Prove that your algorithm is correct.

c. (1p) Analyze the running time of your algorithm.

Solution.

a.
```
ALG(n, A):
    rank[] = [inf] * n
    for judge = 1 ... k:
        i = 0
        for dish in A[judge]:
            if rank[dish] == inf:
                rank[dish] = i + judge
            else if rank[dish] <= i:
                return 0
            i += 1
    return rank
```

b. The algorithm will loop through each judge from judge 1 to judge n. For each judge, it'll check the judge's "local" ranking (the ranking the individual judges gave to the ℓ dishes). The first judge's ranking will be added iteratively (just ranking them in order given the judge). From the second judge onward, the local ranking of the dishes will be checked against the final ranking. If the local rank is greater or equal to the final ranking, it means that the judge had considered the dish worse than a previous judge, thus the rankings disagree with each other, and the algorithm will then return a zero. If there are no problems, then the algorithm will just go on ahead until the last judge, and then return array where the index is the dish number and the value at the index is the dish's ranking (any dishes not judged by any judge will have ranking infinity).

c. Initialization of $rank[]$ is $O(n)$ due to its size. The for-loop will iterate $O(k)$ times, and per iteration the nested for-loop will run $O(\ell)$ times because it'll run through each judge's ranking once. Everything else in the for-loops and out of it are $O(1)$ time. Therefore, the final running time will be $O(n + k\ell)$.

Problem 3 (4 points). Let $G = (V, E)$ be a connected, undirected graph containing n vertices and $n + 5$ edges. Each edge $e \in E$ has a distinct weight $w_e > 0$.

a. (2p) Give an $O(n)$-time algorithm that returns an MST of G.

b. (1p) Prove that your algorithm is correct.

c. (1p) Analyze the running time of your algorithm.

Solution.

a.
```
ALG(G):
    mst[] = [[0,0]] * n  //mst[][1] = parent, mst[][2] = weight
    for each vertex in G:
        for each edge = {u, v} in vertex:
            if (mst[v] == [inf, inf]) or (w(edge) < mst[v][2]):
                mst[v] = [vertex, w(edge)]
    return mst
```

b. The algorithm will check every edge for each vertex. If the destination vertex of the edge hasn't been seen before, then they are added to the $mst[]$ temporarily. Then, every time the vertex finds an edge to a vertex that has already been seen but the weight of that edge is smaller than what is currently in the $mst[]$, then input the new smaller weight. This will make only the smallest values to be inputted in the final $mst[]$. Also, because every edge has distinct weights, there will be no duplicates that will added that could cause a vertex to not be connected to the rest of the graph.

c. Initialization of $mst[]$ costs $O(n)$ time. The for-loop will take $O(n)$ iterations, and per iteration the nested for-loop will cost $O(\#neighbor)$. Thus, the for loops will cost $O(neighbor(1) + neighbor(2) + ... + neighbor(n))$, which costs a total of $O(2m)$ for undirected graphs. The total is then $O(n + 2m)$, but the value of m will always be $n + 5$, so the substituting it will result in $O(n + 2(n + 5)$, which can simplify to $O(n)$.

Problem 4 (3 points). Recall Prim's algorithm for the MST problem: Let s be an arbitrary vertex and initialize a set $S = \{s\}$. Repeat the following $n - 1$ times: add the lightest edge e crossing S to the solution (initially empty), and add the endpoint of e not in S to S.

a. (2p) Give an $O(n^2)$-time implementation of Prim's algorithm in pseudocode. Your program should not create a queue, heap, or any data structure other than an array.

b. (1p) Analyze the running time of your implementation.

Solution.

a.
```
ALG(G):
    S[] = [false] * n
    mst[] = [[inf, inf]] * n //mst[][1] = parent, mst[][2] = weight
    for vertex in G:
        for edge = {u, v} in vertex:
            if (not S[vertex]) and (w(edge) < mst[v][2]):
                mst[v] - [vertex, w(edge)]
        S[vertex] = true
    return mst
```

b. Initialization of $S[]$ and $mst[]$ will each cost $O(n)$ (running total $= O(2n)$). Then, the for-loop will take $O(n)$ iterations, and per iteration the nested for-loop will take $O(\#neighbor)$, therefore the entire nested loops will take $O(neighbor(1)+neighbor(2)+ ... + neighbor(n)) = O(2m)$ for undirected graphs. The worst case scenario for undirected graph in regards to the number of edges (i.e. m) is $n(n - 1)/2$, so the nested loops will have the $O(n(n - 1))$ with substitution. Thus, summing all the values up, the total will be $O(2n + (n^2 - n))$ or simplified down to $O(n^2)$.