HW 6: 1a, b

1a) i. $T(n) = \begin{cases} 1 & n \le 1 \\ 3T(n/3) + n & \text{otherwise} \end{cases}$

ii. $a = 3$, $b = 3$, $c = 1 \rightarrow \log_3(3) = 1$

$T(n) \in \Theta(n \log_3(n))$

iii. Recurrence of standard merge sort: $T_2(n) \in \begin{cases} 1 & n \le 1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$

$T_2(n) \in \Theta(n \log_2(n))$ because $\log_2(2) = 1$

Initially it seems as though the three-way merge sort is better due to the higher base of its log in its runtime. However, since three-way merge has to make comparisons between more subarrays, it is actually worse.

b) i. You would iterate through one array and compare each element to every element in the other array. This would have O(mn).

ii. You would iterate through one array and use a binary search to compare it to every other element in the other array. The iteration would be O(m) and each binary search O(logn).

iii. You would use a method analogous to merge sort. We would pass the two sorted arrays into this method and find their intersection. Since you only iterate through each array once, you get O(m + n).

Scanned with CamScanner

HW 6: 1b, c

1.b) iv. The solution in ii is better for small m and large n due to the log(n) term. However, if m and n are similar in size then the solution in iii. is better because its runtime is equally dependent on m and n.

1.c) Create an output array that's the same size as the array. Since insertion sort doesn't use any additional data structures, it won't add to space complexity. Then iterate through the sorted input array and add the first guest of a certain age to the output array. The time complexity will be $O(n^2)$.

Scanned with CamScanner

HW 6: 2a, b, c, d

2a) i. You should first create K copies of the graph. Then, run Dijkstra's algorithm on a copy of the graph with a person as the source vertex, and do this for all K people. Finally, traverse each graph using the smallest distances found through Dijkstra's algorithm to find the shortest path.

ii. $O(K(|V|+|E|+|V|\log(|V|)+|E|\log(|V|)+|V|+|E|))$

b) i. $O(|V|+|E|)$

ii. A path from Odegaard to some location, say a person.

iii. Run Dijkstra's algorithm on this reversed graph. Then, start at a person node and use its predecessors to trace back to Ode, thus finding the path.

iv. $O(|V|\log(|V|)+|E|\log(|V|)+K|V|)$

c) A pizza delivery company trying to find the fastest route to deliver pizza to multiple houses (assuming each driver starts at the store).

d) i. Create a new node and give it an edge with weight zero directed to an ambulance. Then, add another edge with weight zero connected from the first ambulance to the second ambulance, and another edge from the second to third ambulance with weight 0.

Scanned with CamScanner

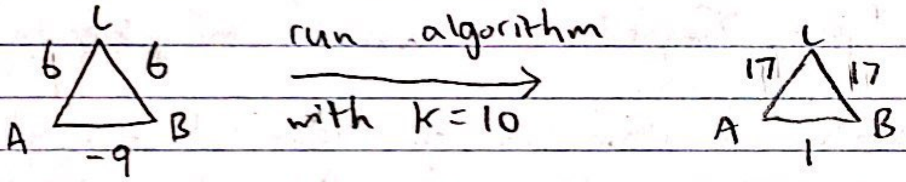
HW 6: 2d, 3a, b

2.d) ii. Check the predecessors of the emergency node until you reach an ambulance node, and that's which ambulance you send.

iii. $O(1)$. Since these are no collisions, modifying and adding to a hash table will be constant time.

e) Uber could use this strategy to send the closest driver to a passenger.

3.a)



The shortest path from A to C before the algorithm is run is A→B then B→C, at a cost of 3. However, after the algorithm is run, the shortest path from A to C is just A→C. As such, after the algorithm is run, the incorrect answer is given.

b) $|E| = |V|^2 - |V|$. We can see this from the fact that each vertex can have two edges with every other vertex. We can plug this into the runtime for Dijkstra's algorithm:

$$O(|V|\log(|V|) + (|V|^2 - |V|)\log(|V|) = \boxed{O(|V|^2\log(|V|)}$$

This leads to $|E| = 2\cdot\sum_{i=1}^{|V|} i = 2\cdot\frac{|V|(|V|-1)}{2} = |V|^2 - |V|$

Scanned with CamScanner

HW. 6: 3c, 4a, b

3. c) This answer only applies to simple graphs because non-simple graphs can have self loops and parallel edges. The addition of self loops will cause the worst-case runtime of a non-simple graph to exceed that of a simple graph.

4a) i. Each vertex will be a location and each edge will be the distance between two vertices.

ii. Each will store the distance between two vertices. Each vertex will store both the shortest distance from the source and the number of vertices on that shortest path.

iii. This will be a weighted graph.

iv. This should be a directed graph so that you can find directions in different directions.

V. There is no need for self-loops or parallel edges as they don't make sense in the context of this problem.

b) i. I would add a field "vert" to the Vertex object to keep track of the vertices on the path. In an if statement in the for loop I would compare vertices on the path in addition to the distance.

Scanned with CamScanner

HW 6 : 4b

4.b)ii. ~~I would modify line 8. As such, I would resolve the tie during the BFS.~~
I would add an if statement after line 10 in the for loop. I would check u.vert +1 < V.vert. If true, V.vert would equal u.vert +1. As such ties would be resolved during the BFS.