Question 1:

Algorithm 1 Optimal revenue placement for gas stations

a)
1: S = list of potential station locations
2: R = list of revenues associated with stations by index
3: d = hash of already computed results

4: **procedure** OPT (INDEX N)
5: **if** $n < 0$ **then** ▷ O(2) - Case to handle empty list as input
6: return 0
7: **if** $j = 0$ **then** ▷ O(2) - Only one element is left
8: return S_0
9: **if** $S_n - S_0 < 20$ **then** ▷ O(3) - No potential candidate stations
10: Find max in $R_{0...n}$, and return index of max value ▷ O(20) at worst

11: **for** i = $n - 1$; i ≥ 0; decrement(i) **do** ▷ O(20) at worst
12: **if** $S[n] - S[i] \geq 20$ **then** ▷ O(3)
13: let c_n = i ▷ O(1) - c_n denotes 1^{st} index at least 20 away from $S[n]$
14: *break* ▷ c_n has been discovered, stop further iterations

15: If function call for an arbitrary $k < n$, $opt(k)$ has been made, do look up in hash, d, instead of recomputing O(1) for look up

16: $d_{c_n} = opt(c_n)$ if d_{c_n} not defined
17: $d_{n-1} = opt(n-1)$ if d_{n-1} not defined
18: return $max(R_n + d_{c_n}, d_{n-1})$ ▷ Either station n is put in opt placement or it isn't
19: $opt(|S| - 1)$ ▷ Begin at the index of the last element

The following was used as a resource for the algorithm above:
https://cgi.csc.liv.ac.uk/ martin/teaching/comp202/Exercises/Dynamic-programming-exercises-solution.pdf

b) Solving any given $opt(n)$, can be shown as in terms of solving the smaller sub problems of $opt(n)$; as at each recursive step, the best option, in terms of revenue, of two smaller sub problems $opt(k)$ are solved.

At every instance of $opt(k)$ for any given $k \leq n$, the value returned would be the max (returns the higher value of the arguments provided), of creating this station (seen as adding the current revenue of the station to the solution), or ignoring this station and moving on to the next one. So the question for any given station becomes is it part of the solution or not.

It can be shown the proof by induction that the solution for the most optimal placement, must be composed of the optimal solutions of a smaller subset of the stations.

Base Cases:

Where n is the number of stations ...

n = 1:
Only one station exists, optimal placement will have to include this station.

n = 2:
Either the distance between these two stations is less than 20 away from each other or not. In the case that they are less than 20, we find the best option of the two by performing *max* on the two stations' revenues.

If they differ by at least 20 in distance, the first station would become c_n of $n = 2$, and the optimal revenue would be finding the max of including both revenues of each station in the solution, or ignoring $n = 2^{nd}$ station, and reducing to base case n=1.

Inductive Hypothesis:
Assume that for $2 \leq k \leq n$, $opt(k)$ for the claim holds true that *opt* returns the optimal station placement. If it can be shown that for an arbitrary $n + 1$ that $opt(n + 1)$ still holds true, then it must also be true for $opt(n)$

Inductive Step:
Adding an $(n + 1)^{th}$ index of a station and a correlated revenue, there are two options to consider. It's either in the solution or not.
In the case that index $n + 1$ is in the solution, i.e its optimal revenue for station placements is greater than the optimal revenue for not placing it, and station $n + 1$ is part of the optimal placement solution. Therefore the solution would become R_{n+1} (the revenue at index n+1) plus $opt(n)$, where the inductive hypothesis can be applied.

If it's not part of the solution, proceed without including it in the optimal placement solution; thus the optimal solution would remain as $opt(n)$, where the optimal solution holds true according to the inductive hypothesis.

It can be seen that the claim will hold for any value of n, since it can be shown for a list of stations of any arbitrary size.

c) In the context of one recursive call, the expense of finding an element to place is constant amount of operations.

There are two properties of the potential station placements that should be stated before continuing. The station locations are discrete, meaning that there can't be an infinitesimal amount of stations. As well as the fact that every station location is unique, meaning no two stations can exist at the same point along the X axis. This allows us to place a constant upper bound on the amount of items we need to check.

In the case that distance between S_n and S_0 is less than 20, we find the max of these elements. Because of the above mentioned properties, that means that there are at most 20 stations to check before we find the best station, in terms of revenues.

The case where S_n and S_0 is greater or equal than 20, we find the first candidate, c_n, where c_n is a station index s.t the distance from c_n and n is at least 20. Again because of the above property, there can be at most 20 stations before we find c_n.

Within the context of one recursive call, the amount of operations is constant. In terms of how many calls to *opt* are made, this is at most n. Since the worst case scenario would be at each step, the next given index is a candidate, c_n of the next element. This can only happen n number of times, where n is the size of the input list. This is assuming no two optimal placements for a given index are computed more than once. This is handled

by memoization. For a call to $opt(k)$ for an arbitrary $k < n$, if the call has already been made, we perform a look up instead of recomputing it. Using a hash as our look up table, this can happen in O(1).

For the sake of simplicity, I'll show the number of constant operations per recursive step to be 20; the maximum number of items needed to be checked.

$$T_n = 20 \cdot T_{n-1} = O(n)$$
$$T_n = 20n \leq cn$$
$$c = 20$$

Let $c = 20$ and $n_0 = 1$. $T(n) \leq cn$, for all values $n \geq n_o$.

Question 2:

Algorithm 2 Longest Common Substring

a)
```
1:  procedure LCS (String A, String B)
2:      M = matrix int[|A| + 1][|B| + 1]
3:      for i = 0; i ≤ |A| do                          ▷ Fill in left column with zeros
4:          M_{i,0} = 0
5:      for i = 0; i ≤ |B| do                          ▷ Fill in top row zeros
6:          M_{0,i} = 0
7:      for i = 1; i ≤ |A| do                          ▷ Building the matrix
8:          for j = 1; j ≤ |B| do
9:              if A_{i-1} == B_{j-1} then
10:                 M_{i,j} = M_{i-1,j-1} + 1          ▷ Substring continues, continue to increment
11:             else
12:                 M_{i,j} = 0                        ▷ Gap has been found, reset the incrementation

13:     let k = r_i = r_j = −∞

14:     for i = 0; i ≤ |A| do                          ▷ Finding the largest matrix coordinate
15:         for j = 0; j ≤ |B| do
16:             if k < M_{i,j} then                    ▷ A longer common substring has been found
17:                 k = M_{i,j}
18:                 r_i = i − M_{i,j} + 1
19:                 r_j = j − M_{i,j} + 1
20:     return r_i, r_j, k
```

The following was used as a resource for the algorithm above:
http://algorithms.tutorialhorizon.com/dynamic-programming-longest-common-substring/

b) First begins the process of initializing the first row and first column to be 0. This allows the incrementation of the next variable to signify a matching of index i in string A, and index j in string B.
The algorithm then proceeds constructing the matrix through nested iteration of A and B. Since we've initialized the first row and column, we will use $i - 1$ and $j - 1$ to show the current index of iteration. If A_{i-1} is equal to B_{j-1}, for any given index i or j, we will set $M_{i,j}$ to be the accumulated sum of the $M_{i-1,j-1}$ (which represents the length of that common substring), plus one to signify its continuation. If at any point there's a

gap, that is to say that if $A_{i-1} \neq B_{j-1}$, the length of that common substring is set to 0. Signifying the end of that substring.

Once the construction has been complete, we find the largest value, and set it to k. If at any point, a larger k is found, we will update our k. The starting, and ending index our also updated whenever a larger k is found.

We can show that this algorithm is correct through contradiction. Assume that LCS returns a string that is not the longest common substring. This cannot ever be the case, since the choosing of the longest substring only occurs after the matrix M has been completely constructed. If there would exist a k, that represents the length of the string, larger than the k returned by LCS, LCS would have return that k instead.

c) The algorithm can be broken into several phases to make run time analysis more simple; initialization, and construction, and look up. The initialization phase is quite simple to analyze, as it consists of filling the top row, and left column of the matrix. This happens in O(n) + O(m), as $n + m$ operations occur during this phase; where n is the size of B, and m the size of A.

As for the construction phase, where the values in the matrix are set to either be the incrementing sum, or zero if ever $A_i \neq B_j$. This phase performs about $n * m$ operations, as every value in the matrix needs to be visited.

Looking up the value, a second nested for-loop in the algorithm will do another $n * m$ operations, iterating through the matrix to find the max value. In truth building the matrix, and looking through it can be done in the same nested for-loop. But for the sake of making it more understandable, it can be split into two different tasks.

In total we have, disregarding simple operations such as assignments, or comparisons:

$$\begin{aligned} T_{m,n} &= m + n + 2mn = O(mn) \\ m + n + 2m \leq m \cdot n + n \cdot m + 2mn &\leq c \cdot mn \\ 4mn &\leq c \cdot mn \end{aligned}$$

Let $c = 4$, $m_0 = 1$, $n_0 = 1$, it becomes evident that $T_{m,n} = O(mn)$ for all values of $n \geq n_0$ and $m \geq m \geq m_0$.

Question 3:

Algorithm 3 Determines if there is path from x to all other vertices in directed G = (V,E)

a)
```
1:  procedure CONNECTED(VERTEX x)
2:      let visited = {}          ▷ Hash signifying if a given vertex has already been visited
3:      for v ∈ V do
4:          visited[v] = false
5:      explore(x)
6:      for v ∈ V do
7:          if visited[v] = false then
8:              return false
9:      return true

10: procedure EXPLORE(VERTEX v)
11:     visited[v] = true
12:     for edge (v, u) ∈ E do
13:         if visited[u] = false then
14:             explore(u)
```

b) This algorithm, given a input directed graph, starting from given vertex $x \in V$, will return true if all other vertices are reachable. Now assume this property is true and that there exists some vertex, $u \in V$ that our algorithm has not visited. Since the algorithm will iterate through all the available edges from any vertex visitable from vertex v, this means one of following:

For any vertex $v \in V$ that has been visited, none of them have a directed edge to vertex u. In this case, our the input directed graph has no path from V_x to V_u and will correctly return false, as their exists a vertex that has not been visited.

The other possibility being that there are no other vertices left to visit, in which case, the algorithm will rightly return true by default.

c) The explore procedure is called once from the context of the connected procedure. In terms of time complexity, the worst case would be if the graph was connected, as all edges would be visited. This gives $|E|$ number of operations.

The connected procedure has to initialize, and then check all the values of the visited hash, having the size of V. Therefore at worst $2|V|$ number of operations will occur in the process of initialization and checking if it contains any false values. The worst case scenario in this portion of the algorithm being that there are no unvisited vertices, as all of the items in visited would have to be checked.

The total resulting in $2|V| + |E|$ number of operations. The run time recurrence can be shown in terms of the number of vertices, i.e the size of the vertex set.

$$
\begin{aligned}
T_{|V|} &= 2|V| + |E| = O\big(|V| + |E|\big) \\
T_{|V|} &= 2|V| + |E| \leq c \cdot \big(|V| + |E|\big) \\
2|V| + |E| &\leq 2|V| + 2|E| \leq c \cdot \big(|V| + |E|\big) \\
2 \cdot \big(|V| + |E|\big) &\leq c \cdot \big(|V| + |E|\big)
\end{aligned}
$$

Let $c = 2$, $|V|_0 = 1$, $|E|_0 = 1$. For any given size of vertex set V, and edge set E greater than or equal to $|V|_0$ and $|E|_0$, this equality will hold; $T_{|V|}$ is $O(|V| + |E|)$.

Question 4:

a) Induction can be performed on the number of edges in the graph to show that it must be true that there are an even number of vertices of odd degree in any given graph G.

Base Case: 1 Edge
An edge shares two endpoints, so no matter how many vertices there are in the graph, only the two adjacent endpoints of this edge are effected; setting their degrees to one. The graph now contains two odd degree vertices; any other potential vertex in the graph will have degree 0. It holds that the graph has an even number of odd degree vertices.

Inductive Hypothesis:
Assume that it holds that a graph with n edges, for any $n \geq 1$, will have an even number of odd degree vertices. It can be shown through induction that for any arbitrarily large number that this holds true.

Inductive Step:
Since an edge shares two vertices we know that this action of removing an edge will only affect the two endpoint vertices of this edge. Let's denote these two vertices as V_u and V_v, and the edge they share as E_{uv}. Before the removal of any arbitrary edge there are several cases in terms of the degree of V_u and V_v before and after the operation occurs.

Upon removal of edge E_{uv}, a few cases need to be considered.

Case 1: V_u, and V_v have even degree before removing E_{uv}:
Deleting E_{uv} will decrease the degree of both vertices by 1. Since they were both of even degree, this action will make both vertices V_u, and V_v to be of odd degree. Thus in total, the graph's count of odd degree vertices will increase by two; therefore still holds that the graph has an even number of odd degree vertices.

Case 2: V_u, and V_v have odd degree before removing E_{uv}:
Deleting E_{uv} will decrease the degree of both vertices by 1. Since they were both of odd degree, this action will make both vertices V_u, and V_v to be even. Thus the count of odd degree vertices in our graph decreases by two, and it still holds that there are an even number of odd vertices.

Case 3: V_u, and V_v have even and odd degree before removing E_{uv}:
Deleting E_{uv} will decrease the degree of both vertices by 1. This means that the odd degree vertice would become even, and vice versa. Thus the count of odd degree vertices will not change, as the net of this transaction is 0 (lose one odd degree vertice, gain one); it still holds that there are an even number of odd vertices.

This same sort of logic can be applied when adding an edge as well. As cases 1 and 2 switch in their affect to the graph, in terms of the number of odd degree vertices after performing the transaction. Case 3 remains the same, the net difference of 0 would stay the same. Thus it can be seen that for any given graph G, there will always be an even number of odd degree vertices, for any arbitrarily small or large $|E| \geq 1$.

b) Two paths are edge disjoint if they don't share an edge. Given a even number ($2k$) of

odd vertices, it can be shown that it's possible to form k edge disjoint paths, if no two paths exist where an edge is shared. Through induction on k, the number of pairs of vertices, it can be seen why it must be true that you can have k edge-disjoint paths:

Base Case: One path exists (k=1)

It can seen that a path exists such that it is edge disjoint, as no other paths exist.

Inductive Hypothesis
Given a graph G, with $k \geq 1$ pairs of odd degree vertices a path exists from P_1 to P_k where all paths between these pairs are edge disjoint.

Inductive Step
Remove any path P, the degree of any vertex along this path decrease by two, as a path goes through that vertex, and registers an in-degree and out-degree, The affect of this action will decrement the overall degree of this vertex by 2. Since we are only concerned with vertices of odd degree, this has no apparent affect, as if the degree was odd it would stay odd; thus we can refer to our inductive hypothesis. If it results in a vertex of even degree, we wouldn't have to concern ourselves with it, as we only care of vertices of odd degree.

Thus it can be seen that through removal of an arbitrary path that the claim will hold for a graph with any k number of odd vertex pairs.

Question 5:

a) The shortest path P_{v_1,v_k} must also be the shortest path of it's sub-portions P_{v_1,v_i}, and P_{v_i,v_k}, as if a shorter path had existed, it would replace a given portion, thus returning a shorter path.

To explicitly show that this is true, one portion will be addressed at a time. Working under the assumption that $P_{v_1,k}$ is the shortest path.

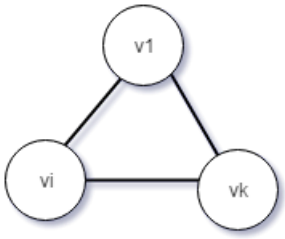
Case 1: Modification of P_{v_1,v_i}, while P_{v_i,v_k} remains static:

If there existed a shorter path from v_1 to v_i, that would contradict the fact that P_{v_1,v_k} is the shortest path, as a shorter path from from v_1 to v_i would exist, and would replace it as P_{v_1,v_i}; as our path from P_{v_i,v_k} would be of the same length, so its clear that a shorter P_{v_1,v_i} would imply there exists a shorter path P_{v_1,v_k}.

Case 1: Modification of P_{v_i,v_k}, while P_{v_1,v_i} remains static:

If there existed a shorter path from v_i to v_k, that would contradict the fact that P_{v_1,v_k} is the shortest path, as a shorter path from from v_i to v_k would exist, and would replace it as P_{v_i,v_k}; as our path from P_{v_1,v_i} would be of the same length, so its clear that a shorter P_{v_i,v_k} would imply there exists a shorter path P_{v_1,v_k}.

b) The same logic applied in a) does not necessarily work when trying to find the longest path; that is to say that P_{v_1,v_i} and P_{v_i,v_k} aren't sub-problems of P_{v_1,v_k}. To show that this is the case, showing an example displaying why it can't be true, would display why P_{v_1,v_i} is **not** the longest path from v_1 to v_i, and P_{v_i,v_k} is not the longest path from v_i to v_k. Take the following illustration as an example:



It can be seen that for v_1 to v_i the longest path is a path through v_k, and that for v_i to v_k, the longest path being through v_1.