Homework 3

CSOR W4231 - Analysis of Algorithms

Chandan Suri

UNI - CS4090

Collaborators: Arnavi Chedda (amc2476), Parth Jawale (pcj2105), Gursifath Bhasin (gb2760), Uday Theja (um2162)

Submitted on -

April 06, 2022

Problem 1

(a)

// Assumption: The coins with denominations are given in an increasing order as a list.
Problem Description:
Input: Given a set of coins of denominations $c_1, c_2, ..., c_n$ which have unlimited supply and the value v for which we need to make the change if possible.
Output: If it's not possible to make the change for the value v provided, then we return No, otherwise we will return a Yes along with the list of coins using which we can create that value.
Solution:
Sub-problem: We try and see if we can create a change based on any of the coins with different denominations. We say that if there is a valid change possible with a coin, then we can move forward and create more change based on the residual left. We create change for values from 0 to v by going over all the denominations. If we can create a change for some value x less than v using the coins, then for $x + 1$ we can check the same and create change. If for some other value y we can create a valid change based on the change created for x, then we can successfully create the change till then. We do this till value v. If at the value v, we can create the changes for some of the residuals and any of the coins, then we can successfully create the change for the value v too. Therefore, the sub-problem is to find whether a value less than v can be made with some coins of some denominations.
Recurrence: As we subtract the value of the coins with some denomination c_i from a value j less than or equal to v. Thus, for all the $j > 0$, we can make a value of j if we can make a value of $j - c_i$ for some $1 <= i <= n$. Thus, we do so for all the values less than v, therefore the recurrence will be as follows:

$$OPT(j) = \max_{1 \leq i \leq n} \{j - c_i \text{ if } j \geq c_i\}$$

Also, we will update the *residual_indices* array with the residual $j - c_i$ value after coin with the denomination from the current value under consideration over which we will backtrack to get the path we took to make the change while I store the denominations in another array at the same value indices denoted by *residual_indices*. As we go over the coins with denominations in an increasing order and break at the first instance where we find a valid change, we are basically maximizing the value of $j - c_i$ implicitly.
Boundary Conditions: As the residuals array will hold the residual after deducting the denomination from a value, for creating a value of 0, nothing can be subtracted which is a valid denomination so the residual will be zero. Also, the array holding the denominations (*denominations_used*) used will have 0 at the 0th index.

$$OPT(0) = 0$$

Order of Array Filling: We will the *residual_indices* array one after the other from 0 to the value v as we are trying to create the change for a sub-problem of the value v. We

also fill the *denominations_used* array with used denomination for a sub-problem value in a similar fashion.

Pseudocode:

Algorithm 1: Make the change from an unlimited supply of coins with some denominations for a value v.

```
1  Function initialize(v):
2      for idx ← 0 to v do
3          denominations_used[idx] ← ∞ ;
4          residual_indices[i] ← −1;
5      end for
6      denominations_used[0] = 0;
7      residual_indices[0] = 0;
8  Function generateChange(coins, v):
9      initialize();
10     for curr_val ← 0 to v do
11         for curr_coin_val ∈ coins do
12             if curr_coin_val ≤ curr_val then
13                 left_over_val = curr_val − curr_coin_val;
14                 if left_over_val > −1 and residual_indices[left_over_val] > −1
                    then
15                     residual_indices[curr_val] = left_over_val;
16                     denominations_used[curr_val] = curr_coin_val;
17                     break;;
18         end for
19     end for
20     if residual_indices[v] == −1 then
21         return False
22     curr_idx = v;
23     changes_used = [];
24     while curr_idx > 0 do
25         if residual_indices[curr_idx] ≠ −1 then
26             changes_used = changes_used ∪ denominations_used[curr_idx];
27             curr_idx = residual_indices[curr_idx];
28         else
29             return False
30     end while
31     return True, changes_used;
```

Time Complexity Analysis: We will do the time complexity analysis for the 2 functions

separately.
For the *initialize* function, we loop from 0 to v and initialize the 2 arrays by assigning values to it. The assignment operations take constant time that are inside the loop as well as outside the loop. For the loop on line 2, the time complexity will be of the order $O(v)$. As other operations are constant time $O(1)$, the time complexity of the *initialize* function will be $O(v)$.
Now, for the *generateChange* function:
Line 9 is calling the initialization function which runs in $O(v)$ time as stated above.
Lines 10 and 11 are two for loops which will run $v + 1$ times and $n + 1$ times respectively, where n is the number of denominations in coins.
Then, all the lines from 12-19 are just conditional and/or assignment statement which run in constant time $O(1)$. Thus, the looping construct involving two for loops run in $O((n + 1)(v + 1)) = O(nv)$ time. Lines 20-23 are just conditional and assignment operations which will run in constant time.
Line 24 is a while loop which will run v times in the worst case. Also, all the lines 25-30 inside the while loop are just conditional, assignment and/or union operations which should run in constant time. Thus, the full while loop construct will run in $O(v)$ time.
Last line 31 is just a return statement which should run in constant time.
Thus, combining the time complexities of all the lines above and ignoring lesser order terms will give us a worst case time complexity of $O(nv)$.
Space Complexity Analysis: Regarding the space considerations for the problem, other than the indices for the loops used, we use 2 arrays for storing the intermediary data and another array for storing the denominations used for making the change if at all it was possible to generate the change for the value given.
Regarding the 2 arrays that we initialize initially:
residual_indices is of the size of the value v for which we need to make the change. And, *denominations_used* is also of the size of the value v which states what denomination was used to create the change for some sub-problem value of v which is denoted by $curr_val$ in the algorithm.
Other than these two arrays, we have another array at the end when it's possible to create a change. This array can hold at most v number of entries in the worst case which gives us final result of a way of creating the change for the value v.
Thus, the total space complexity of the algorithm stated above will be $O(3v) = O(v)$. Conclusively, the total space complexity is of the order of the value v for which the change had to be created. This solution has an optimized space complexity.

(b)

Problem Description:
Input: Given a set of coins of denominations $c_1, c_2, ..., c_n$ and the value v for which we need

to make the change if possible.
Output: If it's not possible to make the change for the value v provided, then we return No, otherwise we will return a Yes along with the set of coins using which we can create that value.
Solution:
Sub-problem: For making the change for a value (less than or equal to the final value v for which the change has to be made), we need to consider two sub-problems. For making the change for a particular value with a denominations, the following 2 sub-problems need to be observed:

1. We were able to make the change for the current value $curr\_val$ with previous coin $denom\_idx - 1$ or denomination. This sub-problem is denoted by $OPT[denom\_idx - 1][curr\_val]$.

2. We were able to make the change for the value $curr\_val - coins[denom\_idx]$ (current value for which the change has to be made - value of the current denomination under consideration) with the previous denomination $denom\_idx - 1$. This sub-problem is denoted by $OPT[denom\_idx - 1][curr\_val - coins[denom\_idx]]$.

Please note that for the pseudocode, as I am following 0-based indexing, for accessing the current denomination value, I have to subtract 1 from it ($coins[denom\_idx - 1]$).
Recurrence: Let $OPT(i, j) = 1$ if the value j can be made using a subset of the denominations $c_1, c_2, ..., c_i$ with no repetitions allowed, and 0 otherwise.
Thus, to make the value j from a subset of the first i denominations, we can either use the i-th denomination or not. Thus, we get the following recurrence:

$$OPT(i, j) = \max \{OPT(i - 1, j), OPT\,(i - 1, j - c_i) \text{ if } j \geq c_i\}$$

Also, if $j \geq c_i$ is not true, then it should be zero if the current value cannot be made using the previous denomination, so, it should be as:

$$OPT(i, j) = OPT(i - 1, j) \text{ if } j < c_i$$

Boundary Conditions: There are two boundary conditions here:

1. $OPT(i, 0) = 1$ for all $0 \leq i \leq n$. This is the case when we can make change for 0 value with any of the denominations.

2. $OPT(0, j) = 1$ for all $0 \leq j \leq v$. This is the case when we cannot make change for any value with no denominations.

Order of Matrix Filling: We fill the OPT matrix here row by row for each of the denominations by going over all the values for which the change can be made either using that denomination or the ones before it (a subset is possible). Basically, the loop takes up

a denomination one by one and for that denomination value, we see whether we can create the change for all the values from 0 to value v.
Pseudocode:

Algorithm 2: Making change for the value v with some denominations (repetition not allowed).

```
1  Function initialize(n, v):
2      OPT = Array(n + 1, v + 1);
3      for row_idx ← 0 to n do
4          for col_idx ← 0 to v do
5              OPT[row_idx][col_idx] ← −1;
6              if row_idx == 0 then
7                  OPT[row_idx][col_idx] ← 0;
8              if col_idx == 0 then
9                  OPT[row_idx][col_idx] ← 1;
10         end for
11     end for
12 Function generateChange(coins, v):
13     num_coins = |coins|;
14     initialize(num_coins, v);
15     for denom_idx ← 1 to num_coins do
16         for curr_val ← 1 to v do
17             if curr_val ≥ coins[denom_idx − 1] then
18                 OPT[denom_idx][curr_val] =
                    max(OPT[denom_idx − 1][curr_val], OPT[denom_idx −
                    1][curr_val − coins[denom_idx − 1]);
19             else
20                 OPT[denom_idx][curr_val] = OPT[denom_idx − 1][curr_val];
21         end for
22     end for
23     if OPT[num_coins][v] == 0 then
24         return False
25     curr_val_idx = v;
26     curr_denom_idx = num_coins;
27     changes_used = [];
28     while curr_val_idx > 0 do
29         if OPT[curr_denom_idx][curr_val_idx] ≠ 0 then
30             changes_used = changes_used ∪ coins[curr_denom_idx − 1];
31             curr_val_idx = curr_val_idx − coins[curr_denom_idx − 1];
32             curr_denom_idx = curr_denom_idx − 1;
33         else
34             return False
35     end while
36     return True, changes_used;
```

Time Complexity Analysis: We will analyze the time complexity of the functions one by one:
For the *initialize* function:
Line 2 is just declaring a 2-D array which is basically a constant time operation.
Lines 3-4 have two for loops that are run $n+1$ times and $v+1$ times respectively. Also, each of the lines 5-11 inside this for loop construct are constant time operations as these are simply conditional, initialization statements. Thus, this full loop construct and hence, the initialization function runs in $O((n+1)(v+1)) = O(nv)$ time.
For the *generateChange* function:
Line 13 is just getting the length of the coins array which is considered a constant time operation. Also, line 14 is calling the *initialize* function which basically runs in $O(nv)$ time. Lines 15-16 has two for loops that are n (number of coins) times and v times respectively. Also, the lines 17-21 are just conditional or assignment operations which are constant time operations. Thus, this full loop construct runs in $O(nv)$ time.
Lines 23-27 are just conditional, return or assignment operations which run in constant time. Line 28 contains a while loop which is run n times in the worst case and all the lines 29-35 are basically conditional, assignment or union operations (1 element at a time) which run in the constant time. Thus, this full while loop construct will run in $O(n)$ time.
Last line 36 is just a return statement which will run in constant time.
Thus, the final time complexity for the whole algorithm can be computed by adding up all the time complexities above and ignoring higher order terms and constant which gives us that the final worst-case time complexity of the algorithm is $O(nv)$.
Space Complexity Analysis: Regarding the space considerations here, other than the indices for the loops, we have a 2-D array OPT of size nv and a final result array naming *changes_used* that can hold at most n values. Thus, the space complexity in this algorithm is $O(nv)$. This can be further optimized but this solution has sufficient enough space complexity of $O(nv)$.

Problem 2

Problem Description:
Input: A graph is constructed with vertices as the currencies $c_1, c_2, ..., c_n$ and edge weights as the exchange rates which was initially a table R with nxn exchange rates.
Output: Returns "Yes" if there exists a subset of currencies $c_{i_1}, c_{i_2}, ..., c_{i_k}$ such that $R[i_1, i_2] \cdot R[i_2, i_3], ..., R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1$, Otherwise "No" is returned suggesting no subset of such currencies exists among the n currencies and nxn exchange rates.

Solution:
Description of the Algorithm:

Initially, we are given the exchange rates for n currencies in the form of a n x n table R such that $R[i, j]! = 0$.For the same, we can construct a graph from this having vertices as the currencies and edge weights as the exchange rates that are provided in the table R.
For all the algorithms discussed in class, we generally deal with adding the edge weights. However, we would need to multiply the edge weights which denote the exchange rates rather than adding them. Using the logarithmic property wherein multiplication of numbers can be converted to addition of those number by taking the log of it. To transform these edge weights denoting exchange rates, we would firstly need to take ln (logarithm) on both sides of the equation in which all the exchange rates (which give a value greater than 1 when multiplied) are multiplied in a cyclic manner to give an arbitrage greater than 1. This computation is shown below:

$$R[i_1, i_2] \cdot R[i_2, i_3], ..., R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1$$
$$ln(R[i_1, i_2] \cdot R[i_2, i_3], ..., R[i_{k-1}, i_k] \cdot R[i_k, i_1]) > ln(1)$$
$$ln(R[i_1, i_2]) + ln(R[i_2, i_3]) + ... + ln(R[i_{k-1}, i_k]) + ln(R[i_k, i_1]) > 0$$

As we are converting the exchange rates and the associated currencies into a graph, it would be beneficial if we can convert this inequality in some way such that we can find this kind of a cycle. As we know some algorithms that can help us find negative cycles in a graph, we can convert the above inequality to give us > 0 value by multiplying both sides by -1. This would convert the LHS of the above inequality into the summation of negative logarithms:

$$-ln(R[i_1, i_2]) - ln(R[i_2, i_3]) - ... - ln(R[i_{k-1}, i_k]) - ln(R[i_k, i_1]) < 0$$

After the above transformation, our problem is converted into a problem where we would need to find whether a negative cycle exists in the graph. If we come across a negative cycle in the graph, we can deduce that a subset of currencies exists for which the product of the associated exchange rates is greater than 1.
Essentially, we begin our algorithm by taking the negative natural logs (ln) of the weights of the edges of the graph and then run the Floyd-Warshall algorithm to fill up the DP table that gives us the distances from every vertex to every other vertex in the graph formed. Then, by accessing the DP table formed by the Floyd-Warshall algorithm, we simply loop through the table only accessing the diagonal elements which gives the distances of all the vertices to itself as for arbitrage the currency when converted back after multiple conversions to other currencies gives us profit.
By the proof of the inequality involving natural logs above we can say that when $D[i][i] < 0$, then a negative cycle exists in this graph suggesting that for that currency we get a value greater than 1, when all the intermediary associated exchange rates for the conversion of currencies are multiplied. Thus, we would return $True(Yes)$ for this case. Otherwise, if $D[i][i] \neq 0$ for all vertices of the graph, then there is no negative cycle present in the graph and we would return $False(No)$.

Pseudocode:

Algorithm 3: Check whether some subset of currencies when multiplied gives a value that's greater than 1.

```
1  Function areCurrenciesGreaterThanOne(G = (V, E, w)):
2      for (u, v) ∈ E do
3          w(u, v) = −ln(w(u, v));
4      end for
5      D = Floyd − Warshall(G = (V, E, w));
6      for curr_idx ←0 to |V| do
7          if D[idx][idx] < 0 then
8              return "Yes";
9      end for
10     return "No";
```

Correctness of the Algorithm:
For proving the correctness of our algorithm, we would need to consider the 3 parts of our algorithm. The 3 parts are:

1. Taking the negative logarithm of all the edge weights that denote the exchange rates.
2. Running the Floyd-Warshall algorithm to get the distances between one vertex and another vertex for all the vertices in the graph.
3. Finding a negative cycle in the graph by looking at the diagonal elements of the table returned by the 2nd part above.

As I am using the Floyd-Warshall algorithm as it is which has been discussed in the class, I will not be proving correctness for the second part above. Thus, I have to prove the correctness for only the 1st and the 3rd part above. The clauses that we need to prove for the parts above are as follows:

1. We will have to prove that the sum of negative natural log of the weights of the graph edges denoting the exchange rates gives us a value that is less than 0 and that this would be sufficient to prove that the product of the exchange rates is greater than 1.
2. We will have to check that the DP table's diagonal (denoting cycles for all the vertices) for a negative value is sufficient to detect the presence of a negative cycle in the graph.

Firstly, for the proof of the first clause above:
Suppose we have 5 currencies as c_1, c_2, c_3, c_4, c_5 which form a directed graph that has a valid

cycle starting and ending at c_1. The edge weights that we will consider here are as follows:

$$\begin{aligned} c_1 \text{ to } c_2 &\implies p \\ c_2 \text{ to } c_3 &\implies q \\ c_3 \text{ to } c_4 &\implies r \\ c_4 \text{ to } c_5 &\implies s \\ c_5 \text{ to } c_1 &\implies t \end{aligned}$$

Now, let's assume that p, q, r, s, and t (denoting the exchange rates) when multiplied gives a value greater than 1. Thus, this can be denoted as follows:
$p \cdot q \cdot r \cdot s \cdot t > 1$,
Taking negative logarithm (ln) on the both sides of the inequality above gives us:

$$\begin{aligned} & ln(p \cdot q \cdot r \cdot s \cdot t) > ln(1) \\ & -1 \cdot (ln(p) + ln(q) + ln(r) + ln(s) + ln(t)) < -1 \cdot ln(1) \\ & -ln(p) + -ln(q) + -ln(r) + -ln(s) + -ln(t) < 0 \\ & -ln(p) - ln(q) - ln(r) - ln(s) - ln(t) < 0 \end{aligned}$$

After analyzing the inequality above, we can say that:

1. If the product $p \cdot q \cdot r \cdot s \cdot t$ is greater than 1, then $-ln(p) - ln(q) - ln(r) - ln(s) - ln(t)$ is going to be negative as shown above. This fits well in line with the fact that when $x > 1$ in $-ln(x)$, then $-ln(x)$ is going to be negative which suggests the presence of a negative cycle. Thus, the product being greater than 1 is equivalent to there being a negative cycle.

2. Conversely, If the product $p \cdot q \cdot r \cdot s$ is lesser than 1, then $-ln(p) - ln(q) - ln(r) - ln(s) - ln(t)$ is going to be positive. This suggests the absence of a negative cycle here.

Thus, taking negative logarithm of the edge weights is both imperative and sufficient here for our algorithm.
Secondly, to prove the presence of a negative cycle based on the table formed by the Floyd-Warshall algorithm. As the Floyd-Warshall algorithm gives the shortest paths between all the vertices in the graph, it also gives us the shortest distance for each vertex from itself along the main diagonal of the table formed. A negative cycle when present, would makes at least one of these distances between the vertex from itself negative. This would suggest the presence of a subset of currencies which when multiplied gives a value greater than 1. Thus, checking for these diagonal values as $D[vertex, vertex] < 0$ should give us the presence of the negative cycle and this is the a sufficient condition for the same.
As we have sufficiently proven the parts of the algorithm, we have proved the correctness of the algorithm.

Time Complexity Analysis: Lines 2-4 takes $O(m)$ time (where m suggests the number of edges), since we loop through every pair of edges in the graph G.
Line 5 calls the Floyd-Warshall routine on the graph G which has a complexity of $O(n^3)$ for a dense graph and this will be a dense graph as $R[i, j]! = 0$ for any i and j vertices.
In lines 6-9 the for loop runs n times for each of the vertices and other operations insides the loop are of constant time which gives us a time complexity for the same as $O(n)$.
Line 10 is a return statement which takes constant $O(1)$ time.
Thus, the total worst case time complexity of the algorithm is $O(n^3)$ where n is the number of currencies or vertices in the graph formed.

Problem 3

(a)

Problem Description:
Input: A directed graph G(V, E, w) with potentially negative edge weights, a destination node t and a set of finite values denoting distances as d_v for every $v \in V$.
Output: Returns True if all the d_v's represent the shortest path from nodes $v \in V$ to the destination node t. Otherwise, we return False.

Solution:
Description of the Algorithm: In the question, one of the main points of contention is that generally we have the source fixed and we calculate the shortest paths to other nodes in the graph. However, in this problem, we have fixed the destination and are varying the source nodes in the graph to calculate the corresponding distances. To follow a similar pattern as taught in the class, we will first transform the graph as a transposed version by going over the edges so that our destination node t would becomes as a source now so, that we can use some insights from the traditional algorithms like Dijkstra's for the same.
Following this, for all the edges in the newly formed graph that has the start node as t, we will look at all the edge weights and the associated distances that has been given to us. Also, for all those edges, we check whether $d_v[v] > d_v[t] + w'(t, v)$ is true. If this is true, then that would mean that we have some other path that is shorter than the distance given to us. This would refute the claim by Bob that the distances d_v's are the shortest paths from node v to the destination t. If this happens even once, then the whole Bob's claim can be refuted. If this condition doesn't become true for any of the associated edges and vertices, then we return True at the end suggesting that Bob's claim was correct, thus, verifying Bob's claim. Please note that we only go over the nodes that start with the destination node t in the newly formed graph (now acting as source) for checking the shortest path distances denoted by the set of finite values d_v.

Pseudocode:

Algorithm 4: Verify whether d_v is the weight of the shortest path from the vertex v to the destination vertex t.

```
1  Function transpose(G = (V, E, w)):
2      for (u, v) ∈ G(E) do
3          E' = E' ∪ (v, u));
4          w'_vu ← w_uv;
5      end for
6      G' = (V, E', w');
7      return G'
8  Function update(t, v, d_v):
9      if d_v[v] > d_v[t] + w'(t, v) then
10         return False;
11     return True;
12 Function verifyShortestPaths(G = (V, E, w), t, d_v):
13     G' = transpose(G);
14     for (t, v) ∈G(E') do
15         is_shortest_path_to_v = update(t, v, d_v);
16         if is_shortest_path_to_v == False then
17             return False;
18     end for
19     return True;
```

Correctness of the Algorithm:
For proving the correctness of the algorithm , we would need to prove different parts of the algorithm first.
Firstly, the *transpose* function just reverses the directions of the edges without changing the edge weights which would essentially not change the shortest paths between 2 vertices in the graph and just gets us a new graph with reversed edges to follow uniformity with the algorithms taught in the class.
Secondly, the *update* function is the same condition that is used in Dijkstra's algorithm for updating the distances. As here we aren't updating any distances and are just checking whether there's a potential to change the shortest paths between different sets of vertices, this should run just fine.
Lastly, for the main function $verifyShortestPaths$, after getting the transposed graph, we are just going over all the edges from the destination node t (now treating it as a source node) and checking whether there is a potential to change the shortest distances provided. As this is done exhaustively for all the edges in the graph (new one) looking for a distance

update from the distances (shortest paths) given, this should be correct as well.
Conclusively, we have proven the correctness of the algorithm. Hence, proved!
Time Complexity Analysis: We will analyze the worst-case time complexity here by analyzing each of the functions in the algorithm as shown below:
For the *update* function: As this only contains conditional and return statements (lines 9-11) which are only constant time operations. Thus, this function should run in $O(1)$ time.
For the *transpose* function: We go over all the edges in our graph and flip the direction of edges keeping the weights intact so that we don't change the shortest distances in the original graph. The for loop in line 2 would run n + m times where n denotes the number of vertices and m denotes the number of edges and others are just assignment operations (lines 3-6) which would run in constant time. Thus, this function should run in $O(n + m)$ time.
For the $verifyShortestPaths$ function: We first call the *transpose* subroutine to get the transposed version of the graph which runs in $O(n + m)time$ as stated above. The for loop would run for all the edges starting with the node t and if it's a dense graph, then this for loop would run m times. Also, other statements inside the loop (lines 15-18) are all constant time operations which would have the worst case time complexity as $O(m)$. Thus, the worst case time complexity of the whole algorithm is of the order $O(n + m)$.

(b)

Problem Description:
Input: A directed graph G(V, E, w) with potentially negative edge weights, a destination node t, new destination node t', and a set of finite values denoting distances as d_v for every $v \in V$ which denotes the shortest paths.
Output: Returns new weights d'(v) of the shortest path to all nodes v in the graph to new destination node t'.

Solution:
Description of the Algorithm: The explanation for the *transpose* function would remain as in the part a.
Other than that, $w'_{tv} = w'_{tv} + d_v(t) - d_v(v)$ is the equation used to reweigh the weights in the graph. Once that is done, we run Dijkstra's algorithm to compute the shortest path. After that, we reweigh the graph using the new distances found. This is done to ensure that we go back to the weighting scheme that we had before.

Pseudocode:

Algorithm 5: Determine new weights for shortest path to new destination node t'.

```
1  Function transpose(G = (V, E, w)):
2      for (u, v) ∈ G(E) do
3          E' = E' ∪ (v, u));
4          w'_vu ← w_uv;
5      end for
6      G' = (V, E', w');
7      return G'
8  Function findNewShortestDistanceWeights(G = (V, E, w), t, t', d_v):
9      G' = transpose(G);
10     for (t, v) ∈ G'(E') do
11         w'_tv = w'_tv + d_v(t) − d_v(v);
12     end for
13     d[v] = Dijkstra(G', t');
14     for (t', v) ∈ G'(E') do
15         w'_t'v = w'_t'v − d(t') + d(v);
16     end for
17     return G'(w');
```

Correctness of the Algorithm:
The transpose function works as explained in the a part above. Regarding the function $findNewShortestDistanceWeights$ here, after taking the transpose, I adjust the weights by deducting and adding the shortest distances to source and destination nodes which essentially adds just the amount that is needed to create non-negative weights. By doing this, we will get non-negative weights for sure. As Dijkstra's runs well with non-negative weights, the Dijkstra algorithm will give us the shortest paths to the new destination node t'.
Following this, we readjust the weights using the similar formula with reversed signs so, that we can get the weights in proportion to what we had before. As this is just reversing what we did in the first for-loop, this is supposed to run correctly as well. This proves that our algorithm will run correctly.
So, we have proven the correctness of the algorithm successfully.
Time Complexity Analysis: For analyzing the time complexity of the algorithm, we will have to analyze the functions used separately and then combine the results.
The transpose function will run in $O(n+m)$ time as proven in part a of this problem.
Then, in $findNewShortestDistanceWeights$ function, line 9 calls the *transpose* subroutine which will run in $O(n+m)$ time. Then, the for loops at line 10 and line 14 runs in (m) time in worst case. Other statements in the lines 11 and 13 are simply assignment operations which takes constant time. Also, the Dijkstra's algorithm will run in $O((n+m)logn)$ as taught in the class. Thus, the total worst case time complexity of the algorithm would

become $O((n+m)logn)$. In case of dense graph where $m >> n$, we can neglect the n term in the complexity that was added with m. Thus, the worst case time complexity will become $O(mlogn)$.

Problem 4

Given:
There are two disjoint subsets of size n, S and T, of a finite universe U. A random subset $R \subseteq U$ is sampled by independently sampling each element from the universe U with a probability of p. A sample is considered to be good if it contains an element of T but no element of S. This has been provided to us in the question.
To Be Proven:
When $p = \frac{1}{n}$ (given), the probability that our sample is good is larger than some positive constant that is independent of n.

Solution:
Let S and T be two disjoint subsets of size n such that:
$S \subseteq U$, $T \subseteq U$, $S \cap T = \phi$, U, S and T are finite sets, $|S| = |T| = n > 1$, If $|U| = N$, then $|U| - |S| - |T| = N - 2n$.
Let $R \subseteq U$ such that we randomly sample elements from U following the given rules for a good sample as given above.
Then, let A be the number of elements (elements that are sampled) from set T which are included in R.
Also, let B be the number of elements (elements that are sampled) from set S which are included in R.
Lastly, let C be the number of elements from $(S \cup T)^c = S^c \cap T^c$ be which are included in R.
Furthermore, since each selection of element for set R is generated independently, we can say that S, T and $S^c \cap T^c$ are disjoint partitions of universe U. It can be said that A, B and C are independent as well. From above, we can conclude that each of these will be binomial distributions as stated below:

1. A $\sim$ Bin(n, p)
2. B $\sim$ Bin(n, p)
3. C $\sim$ Bin($N - 2n$, p)

$$\begin{aligned} P(R\,is\,good) &= P(A \geq 1, B = 0, C \in \{0, 1, 2, .., N - 2n\}) \\ &= P(A \geq 1) \cdot P(B = 0) \cdot P(C \in \{0, 1, .., N - 2n\}) \end{aligned}$$

Since we will at least sample something or nothing from the set $S^c \cap T^c$, the $P(C \in \{0, 1, .., N - 2n\}) = 1$
Also, the probability of sample at least one element from T will be equivalent to 1 - probability of sampling nothing from T.

$$\begin{aligned} P(R\,is\,good) &= P(A \geq 1) \cdot P(B = 0) \cdot P(C \in \{0, 1, .., N - 2n\}) \\ &= (1 - P(A = 0)) \cdot P(B = 0) \cdot 1 \end{aligned}$$

Using the formulation for binomial distribution (for probabilities),

$$\begin{aligned} P(R\,is\,good) &= (1 - P(A = 0)) \cdot P(B = 0) \cdot 1 \\ &= (1 - \binom{n}{0} \cdot p^0 \cdot (1 - p)^n) \cdot (\binom{n}{0} p^0 \cdot (1 - p)^n) \\ &= (1 - p)^n \cdot (1 - (1 - p)^n) \end{aligned}$$

Now, we will replace probability p as $p = \frac{1}{n}$ in the above equation:

$$P(R\,is\,good) = (1 - \frac{1}{n})^n \cdot (1 - (1 - \frac{1}{n})^n) \tag{1}$$

Given that:

$$e^x \cdot \left(1 - \frac{x^2}{n}\right) \leq \left(1 + \frac{x}{n}\right)^n \leq e^x$$

We will make the assignment as $x = -1$ in the equation above:

$$(e^{-1} \cdot (1 - \frac{1}{n})) \leq (1 - \frac{1}{n})^n \leq e^{-1} \tag{2}$$

Considering only the latter half of the inequality (2) and multiplying -1 on both the sides (changes the sign of the inequality) and adding one on both the sides, we will get:

$$-(1 - \frac{1}{n})^n \geq -(e^{-1}) \implies 1 - (1 - \frac{1}{n})^n \geq 1 - (e^{-1}) \tag{3}$$

Now, considering only the first half of the inequality (2):

$$(e^{-1} \cdot (1 - \frac{1}{n})) \leq (1 - \frac{1}{n})^n \implies (1 - \frac{1}{n})^n \geq (e^{-1} \cdot (1 - \frac{1}{n}))$$

As we're interested in large value n, we will say that $n \to \infty$:
Using this, we'll get the lower bound of the term we're interested in. Also, by the transitive property, the lower bound of $(e^{-1} \cdot (1 - \frac{1}{n}))$ will also be the lower bound of the term we're interested in, i.e., $(1 - \frac{1}{n})^n$.
Henceforth,

$$(1 - \frac{1}{n})^n \geq (e^{-1} \cdot (1 - \frac{1}{n})) \geq (e^{-1} \cdot (1 - \frac{1}{\infty})) \implies (1 - \frac{1}{n})^n \geq (e^{-1} \cdot (1 - \frac{1}{n})) \geq e^{-1}$$

Thus, we get:

$$(1 - \frac{1}{n})^n \geq e^{-1} \tag{4}$$

Now, using (3) and (4) in (1), we will get:

$$\begin{aligned} P(R\,is\,good) &= (1 - \frac{1}{n})^n \cdot (1 - (1 - \frac{1}{n})^n) \\ &\geq e^{-1} \cdot (1 - (e^{-1})) \\ &\geq \frac{e-1}{e^2}, \end{aligned}$$

As we can see above, the term in the RHS of the inequality is independent of n (a constant). Thus, the probability that our sample is good is larger than some positive constant.
Hence proved!

Problem 5

Given:
Suppose that time is divided in discrete steps. There are n people and a single shared computer than can only be accessed by one person in a single step: if two or more people attempt to access the computer at the same step, then everybody is "locked" during that step. At every step, each of the n people attempts to access the computer with probability p.

(a)

Solution:
For a fixed person i to access the computer, we can say with certitude that other $n-1$ people will not be able to access the computer. It is given that if even one of these $n-1$ people is able to access the computer along with person i, then everybody will be "locked out" of the system.

We know that, probability to attempt to gain access to the computer $= p$
From this, we would know that the probability to not attempt to gain access to the computer is $1 - p$.
Now, the probability that a fixed person i succeeds in accessing the computer during a specific step which would happen when all other $n - 1$ people does not access to the single resource of computer $= p \cdot (1 - p)^{n-1}$

(b)

Solution:
To maximize the above probability, we will differentiate the expression derived in Part (a) with respect to p and set the expression to 0.
Then, we will try to find out the value of p which results in the maximum value of above probability.

$$func(p) = p \cdot (1-p)^{n-1}$$

Let's do it,

$$\frac{d(func(p))}{dp} = 0$$
$$\frac{d[p \cdot (1-p)^{n-1}]}{dp} = 0$$

We will solve for the LHS of the above equation first,
Using the product rule: $\frac{d(uv)}{dx} = u\frac{dv}{dx} + v\frac{du}{dx}$ and chain rule for the 1st term, we get:

$$\implies p\frac{d(1-p)^{n-1}}{dp} + (1-p)^{n-1}\frac{dp}{dp}$$
$$\implies -p \cdot (n-1) \cdot (1-p)^{n-2} + (1-p)^{n-1}$$
$$\implies (1-p)^{n-2} \cdot (-p \cdot (n-1) + (1-p))$$
$$\implies (1-p)^{n-2} \cdot (-n \cdot p + p + 1 - p)$$
$$\implies (1-p)^{n-2} \cdot (1 - n \cdot p)$$

Now, we will equate the above equation to zero as we want to see for what p the probability will be maximized,

$$(1-p)^{n-2} \cdot (1 - n \cdot p) = 0$$

This equation can hold only when the second term of the expression above equals 0 as p won't be equal to 1. Thus,

$$(1 - n \cdot p) = 0$$

Now, we will move the terms around in the equation above,

$$1 = n \cdot p$$
$$p = \frac{1}{n}$$

Thus, for $p = \frac{1}{n}$, the probability that a fixed person i succeeds in accessing the computer during a specific step is maximized.

(c)

Solution:
Now, we have to upper bound the probability such that person i did not succeed to access the computer in any of the first $t = en$ step.

We know that, probability that person i did not succeed to gain access to the computer, $p(failure) = 1 - p(\text{person } i \text{ is able to access computer})$.

$$p(failure) = 1 - p \cdot (1-p)^{n-1}$$

For t steps, this can be written as:

$$p(failure\ in\ any\ of\ the\ t = en\ steps) = (1 - p \cdot (1-p)^{n-1})^t$$

Now, putting $p = \frac{1}{n}$ and $t = en$ in the above equation, we get

$$p(failure\ in\ any\ of\ the\ t = en\ steps) = \left(1 - \frac{1}{n} \cdot \left(1 - \left(\frac{1}{n}\right)^{n-1}\right)\right)^{en}$$

It is given that:

$$e^x \cdot \left(1 - \frac{x^2}{n}\right) \leq \left(1 + \frac{x}{n}\right)^n \leq e^x$$

Putting $x = -1$ and $p = \frac{1}{n}$ in the above inequality and considering the first half of the inequality, we get:

$$\left(e^{-1} \cdot \left(1 - \frac{1}{n}\right)\right) \leq \left(1 - \frac{1}{n}\right)^n \leq e^{-1}$$

$$\Rightarrow \left(e^{-1} \cdot (1 - \frac{1}{n})\right) \leq \left(1 - \frac{1}{n}\right)^n$$
$$\Rightarrow e^{-1} \leq \left(1 - \frac{1}{n}\right)^{n-1}$$

Now, we will divide both sides by n such that the inequality sign doesn't change,

$$\Rightarrow \frac{1}{ne} \leq \frac{1}{n} \cdot \left(1 - \frac{1}{n}\right)^{n-1}$$

Now, we will multiply both sides by -1 and add 1 to both the sides, the inequality sign will change here,

$$\Rightarrow -\frac{1}{ne} \geq -\frac{1}{n} \cdot \left(1 - \frac{1}{n}\right)^{n-1}$$
$$\Rightarrow 1 - \frac{1}{ne} \geq 1 - \frac{1}{n} \cdot \left(1 - \frac{1}{n}\right)^{n-1}$$
$$\Rightarrow \left(1 - \frac{1}{n} \cdot \left(1 - \frac{1}{n}\right)^{n-1}\right)^{en} \leq \left(1 - \frac{1}{ne}\right)^{en}$$

Thus,

$$p(\textit{person } i \textit{ did not succeed in any of the } t = en \textit{ steps}) \leq \left(1 - \frac{1}{ne}\right)^{\lceil en \rceil} \leq \left(1 - \frac{1}{ne}\right)^{en} \leq \frac{1}{e}$$

Thus, the required upper bound is $\frac{1}{e}$.

(d)

Solution:
Now, we need to find the number of steps t required so that the probability that person i did not succeed to access the computer in any of the first t steps is upper bounded by an inverse polynomial in n.

$$\left(1 - \frac{1}{n}\left(\frac{n-1}{n}\right)^{n-1}\right)^{t} \leqslant \frac{1}{n^k}$$

Using the equation given to us, we get:

$$\left(1 - \frac{1}{n}\right)^{n} \leq \frac{1}{e}$$
$$\frac{1}{n-1}\left(1 - \frac{1}{n}\right)^{n} \leq \left(1 - \frac{1}{n}\right)^{n} \leq \frac{1}{e}$$

Using the transitivity property,

$$\frac{1}{n-1}\left(1 - \frac{1}{n}\right)^{n} \leq \frac{1}{e}$$

Now, we multiply both sides by -1 and add 1 to both the sides, this changes the sign of the inequality,

$$-\frac{1}{n-1}\left(1-\frac{1}{n}\right)^n \geqslant -\frac{1}{e}$$
$$1-\frac{1}{n-1}\left(1-\frac{1}{n}\right)^n \geqslant 1-\frac{1}{e}$$

As both the sides are positive, raising them to a positive power of t won't change the sign of inequality,

$$\left(1-\frac{1}{n-1}\left(1-\frac{1}{n}\right)^n\right)^t \geqslant \left(1-\frac{1}{e}\right)^t$$

Putting this with the equation from where had started with this proof,

$$\left(1-\frac{1}{e}\right)^t \leqslant \left(1-\frac{1}{n-1}\left(1-\frac{1}{n}\right)^n\right)^t \leqslant \frac{1}{n^k}$$

Using the transitive property,

$$\left(1-\frac{1}{e}\right)^t \leq \frac{1}{n^k}$$

Take ln on both sides (This will change the sign of the inequality as taking the log of a value that's less than 1 gives a negative output),

$$t \cdot ln\left(1-\frac{1}{e}\right) \geqslant -k \cdot ln(n)$$

Rewriting the equation,

$$t \geqslant \frac{-k \cdot ln(n)}{ln\left(\frac{e-1}{e}\right)}$$
$$t \geqslant \frac{-k \cdot ln(n)}{ln(e-1)-1}$$

Taking negative sign out from LHS's denominator that will cancel out the negative sign,

$$t \geqslant \frac{-k \cdot ln(n)}{-1(1-ln(e-1))}$$
$$t \geqslant \frac{k \cdot ln(n)}{1-ln(e-1)}$$

$$\text{Thus, the number of steps required } = \frac{k \cdot ln(n)}{ln(e-1)-1}$$

(e)

Solution:
Finally, we need to determine the number of steps that are required to guarantee that all people succeeded to access the computer with probability at least $1 - \frac{1}{n}$.

First, Let's take the complement of what we need to determine.
Let C be the event that some people are not able to access the computer. Let t be the steps after which this process fails. Our goal is to minimize the value of t such that the probability of event C happening is very little.

Let $F(i,t)$ be the event that person i is not able to access the computer in any of the 1 through t steps.

Event C will occur only when at least one of the events $F(i,t)$ occurs. This can be expressed as:

$$E = \bigcup_{i=1}^{n} F(i,t)$$

Now, from probability theory, I know that probability of union bounds can be expressed as the sum of probabilities of the event $F(i,t)$,

$$P(\bigcup_{i=1}^{n} F(i,t)) \leqslant \sum_{i=1}^{n} P(F(i,t))$$

If we choose $t = \lceil en \rceil \cdot (c \cdot log_e n)$ and solve the equation above, we get:

$$P(E) \leqslant \sum_{i=1}^{n} P(F(i,t)) \leqslant n \cdot n^{-2} = n^{-1}$$

$$t = 2\lceil en \rceil \log_{\mathrm{e}} \mathrm{n} \text{ rounds}$$