Q1

a)

Let OPT(x, y) be the maximum likelihood of a chain ending at pixel (x, y).

Bellman equation:

$$\text{opt}[x,y] = \begin{cases} l(x,y) & \text{if } y = 0 \\ \max(\text{opt}(x,y-1), \text{opt}(x+1,y-1)) + l(x,y) & \text{if } x = 0 \;\; y > 0 \\ \max(\text{opt}(x-1,y-1), \text{opt}(x,y-1), \text{opt}(x+1,y-1)) + l(x,y) & \text{if } 0 < x < m-1 \;\; y > 0 \\ \max(\text{opt}(x-1,y-1), \text{opt}(x,y-1)) + l(x,y) & \text{if } x = m-1 > 0 \;\; y > 0 \end{cases}$$

It is correct because:

Base case: if y = 0 and $0 \leq x \leq m-1$ meaning we assume the hair ends at the first row and thus is only 1 pixel long, and the maximum likelihood of 1 pixel is the likelihood of the pixel $l(x,y)$

Recursive step:

for $0 < y \leq n-1$, and $0 \leq x \leq m-1$, we assume hair ends at (x, y), we find which direction the hair points to, either north west (x - 1, y - 1), or north (x, y - 1), or north east (x + 1, y - 1) would maximize the likelihood of hair ending at (x, y) and calculate the likelihood.

Initial conditon: if y = 0 opt(x,y) = $l(x,y)$

Top down implementation:

Allocate a matrix M[0, ..., m - 1][0, ..., n - 1]

likelihood(x, y):
    if M[x][y] uninitialized:
        if y = 0:
            M[x][y] = $l(x,y)$
        else:
            if x = 0:
                M[x][y] = max(likelihood$(x,y-1)$, likelihood$(x+1,y-1)$) + $l(x,y)$
            else if $0 < x < m-1$:
                M[x][y] = max(likelihood$(x-1,y-1)$, likelihood$(x,y-1)$,
                        likelihood$(x+1,y-1)$) +$l(x,y)$
            else if x = $m-1$:
                M[x][y] = max(likelihood$(x-1,y-1)$, computeLikelihood$(x,y-1)$) + $l(x,y)$
    return M[x][y]

findHair(P):

```
    allocate M[0, ..., m - 1][0, ..., n - 1]
    run likelihood(0, n - 1), likelihood(1, n - 1), ..., likelihood(m - 1, n - 1)
    maxLikelihood = max(M[0][n - 1], M[1][n - 1], ..., M[m - 1][n - 1])
```

Running time:
Since for each pixel (x, y), in the worst case, a comparison between three numbers is done, then the time complexity for computing each M[x][y] is constant. Since we compute each M[x][y] only once, the total time complexity for computing M[0, ..., m - 1][0, ..., n - 1] is O(mn).

Space complexity:
M[0, ..., m - 1][0, ..., n - 1] is the only space allocated for this algorithm, therefore the space complexity is O(mn).

Order of bottom-up implementation:
We would compute the quantities in the ascending row order. Compute row 1 = {M[0][0], M[1][0], ..., M[m - 1][0]} first, then row 2 = {M[0][1], M[1][1], ..., M[m - 1][1]}, ..., then row n = {M[0][n -1], M[1][n - 1], ..., M[m - 1][n - 1]}. This order works because for row 1, any hair ends at row 1 is 1 pixel long, therefore we can directly calculate the value $l(i, 0)$ for each entry M[i][0] $0 \leq i \leq m - 1$. For any other rows, row j, $0 < j \leq n - 1$, we need information from row j - 1. And in the ascending order they are all calculated and stored in M[0, ..., m - 1][j - 1] and the time complexity for accessing them is O(1)

b)

```
Allocate a matrix Len[0, ..., m - 1][0, ..., n - 1]
shortestPath(x, y):
    if Len[x][y] uninitialized:
        if y = 0:
            if l(x, y) = 0
                Len[x][y] = 1
            else:
                Len[x][y] = 0
        else:
            if x = 0:
                maxL = max(M[x][y - 1], M[x + 1][y - 1])
                if M[x][y - 1] = maxL:
                    if M[x][y - 1] = maxLikelihood:
                    // The hair has already ended
```

```
            Len[x][y] = shortestPath(x, y - 1)
        else if M[x][y - 1] = 0 and l(x, y) = 0:
        // The hair has not started yet
            Len[x][y] = 0
        else:
            Len[x][y] = shortestPath(x, y - 1) + 1
    if M[x + 1][y - 1] = maxL:
        if M[x + 1][y - 1] = maxLikelihood:
            Len[x][y] = min(shortestPath(x + 1, y - 1), Len[x][y])
        else if M[x + 1][y - 1] = 0 and l(x, y) = 0:
            Len[x][y] = 0
        else:
            Len[x][y] = min(shortestPath(x + 1, y - 1) + 1, Len[x][y])
if 0 < x < m - 1:
    maxL = max(M[x - 1][y - 1], M[x][y - 1], M[x + 1][y - 1])
    if M[x][y - 1] = maxL:
        if M[x][y - 1] = maxLikelihood:
            Len[x][y] = shortestPath(x, y - 1)
        else if M[x][y - 1] = 0 and l(x, y) = 0:
            Len[x][y] = 0
        else:
            Len[x][y] = shortestPath(x, y - 1) + 1
    if M[x + 1][y - 1] = maxL:
        if M[x + 1][y - 1] = maxLikelihood:
            Len[x][y] = min(shortestPath(x + 1, y - 1), Len[x][y])
        else if M[x + 1][y - 1] = 0 and l(x, y) = 0:
            Len[x][y] = 0
        else:
            Len[x][y] = min(shortestPath(x + 1, y - 1) + 1, Len[x][y])
    if M[x - 1][y - 1] = maxL:
        if M[x - 1][y - 1] = maxLikelihood:
            Len[x][y] = min(shortestPath(x - 1, y - 1), Len[x][y])
        else if M[x - 1][y - 1] = 0 and l(x, y) = 0:
            Len[x][y] = 0
        else:
            Len[x][y] = min(shortestPath(x - 1, y - 1) + 1, Len[x][y])
if x = m - 1:
    maxL = max(M[x - 1][y - 1], M[x][y - 1])
    if M[x][y - 1] = maxL:
```

```
                if M[x][y - 1] = maxLikelihood:
                    Len[x][y] = shortestPath(x, y - 1)
                else if M[x][y - 1] = 0 and l(x, y) = 0:
                    Len[x][y] = 0
                else:
                    Len[x][y] = shortestPath(x, y - 1) + 1
            if M[x - 1][y - 1] = maxL:
                if M[x - 1][y - 1] = maxLikelihood:
                    Len[x][y] = min(shortestPath(x - 1, y - 1), Len[x][y])
                else if M[x - 1][y - 1] = 0 and l(x, y) = 0:
                    Len[x][y] = 0
                else:
                    Len[x][y] = min(shortestPath(x - 1, y - 1) + 1, Len[x][y])
    return L[x][y]

findHair(P):
    allocate M[0, ..., m - 1][0, ..., n - 1]
    run likelihood(0, n - 1), likelihood(1, n - 1), ..., likelihood(m - 1, n - 1)
    maxLikelihood = max(M[0][n - 1], M[1][n - 1], ..., M[m - 1][n - 1])
    maxLikelihoodCoord= [ ]
    for i = 0 to m - 1:
        if M[i][n - 1] = maxLikelihood
            maxLikelihoodCoord.append(i)
    allocate Len[0, ..., m - 1][0, ..., n - 1]
    L = maxLikelihoodCoord.length - 1
    for k = 0 to L:
        shortestPath(maxLikelihoodCoord[i], n - 1)
     minLen = min(Len[maxLikelihoodCoord[0]][n - 1],...,Len[maxLikelihoodCoord[L]][n -
1])
    let x = -1 // x coord of the end point with max likelihood and shortest path
    for i = 0 to L:
        if Len[maxLikelihoodCoord[i]][n - 1] = minLen:
            x = i
    desiredChain = [ ]
    let y = n - 1
    for y = n - 1 to 1:
        if M[x][y] = maxLikelihood and l(x, y) = 0:
            // Chain has already ended
            continue
```

if M[x][y] = 0 and $l(x, y) = 0$:
    // Chain has not started
    continue
if M[x][y] = maxLikelihood and $l(x, y) \neq 0$:
    // Last point of the chain
    disiredChain.append(p(i, y))
if x = 0:
    find i such that M[i][y - 1] = max(M[x][y - 1], M[x + 1][y - 1]) and if there is a tie choose the one with min Len[i][y - 1]
if 0 < x < m - 1:
    find i such that M[i][y - 1] = max(M[x - 1][y - 1], M[x][y - 1], M[x + 1][y - 1]) and if there is a tie choose the one with min Len[i][y - 1]
if x = m - 1:
    find i such that M[i][y - 1] = max(M[x - 1][y - 1], M[x][y - 1]) and if there is a tie choose the one with min Len[i][y - 1]
disiredChain.append(p(i, y - 1))
x = i
return desiredChain

c)
In our opinion the likelihood funcitons $l(i, j)$ and $l(C)$ are not well designed. Suppose in an image I the hair is 3 pixels long and the pixels are p(i, j), p(i, j + 1), p(i, j + 2) (suppose this chain is chain 1), and $l(i, j) = l(i, j + 1) = l(i, j + 2) =$ a > 0. However the algorithm finds another chain 2: p(s, t), p(s, t + 1), p(s, t + 2), p(s, t + 3), p(s, t + 4), p(s, t + 5), p(s, t + 6) such that $l(s, t) = l(s, t + 1) =$ a, $l(s, t + 6) =$ b > a, and $l(s, t + 2) = l(s, t + 3) = l(s, t + 4) = l(s, t + 5) = 0$. In this case, chain 2 might be caused by some short black chain and a very black dot far away. Because l(chain 2) > l(chain 1), the algorithm falsely decides that the second chain is the hair.

We think modify $l(i, j)$ such that $l(i, j) = 1 - \frac{P(i,j)}{h}$ and keep $l(C)$ the way it is might be better. We punish points that are too bright in the chain.

Q2
a)

$$\sum_{m=0}^{k-1} E[i_m, i_{m+1}] + k * C$$
$$= \sum_{m=0}^{k-1} \sum_{l=i_m+1}^{i_{m+1}-1} |y_l - \hat{y}_l| + k * C$$
where $\hat{y}_l = \frac{y_{i_{m+1}} - y_{i_m}}{x_{i_{m+1}} - x_{i_m}} * (x_l - x_{i_m}) + y_{i_m}$

b)

Step One: Define OPT(i)

OPT(i): the smallest cost to approximate $(x_0, y_0), ..., (x_i, y_i)$

Step Two: Bellman equation

$$OPT(i) = \begin{cases} 0, & i = 0, \\ \min_{0 \leq j < i}\{OPT(j) + E[j, i] + C\}, & i > 0. \end{cases}$$

It is correct becasue:

Base case: if i = 0, meaning there is only one point, no need to do approximation, therefore total error is 0.

Recursive Step: if i > 0, we find which point previous of point p_i that point p_i connects to minimizes the total error and compute the value of the total error, which is the sum of, suppose the previous point is j, cost and error introduced by line segment joining p_i and p_j, which is $C + E[j, i]$, and the total error caused by approximation between point p_0 and p_j, which is OPT[j].

Step Three: Initial conditions

When $i = 0$ $OPT(i) = 0$.

Step Four: top down implementation:

Allocate empty array M and I each with length n + 1

// M[i] for storing smallest cost to approximate $(x_0, y_0), ..., (x_i, y_i)$, I[i] for storing which

//point previous p_i that p_i connects to minimizes the cost

```
minError(j):
    if M[j] not initialized:
        if j = 0
            M[j] = 0
            I[j] = 0
        else:
            min = ∞
            index = -1
            for i = 0 to j - 1:
                cur = minError(i) + E[i, j] + C
                if cur < min:
                    min = cur
                    index = i
            M[j] = min
            I[j] = index
```

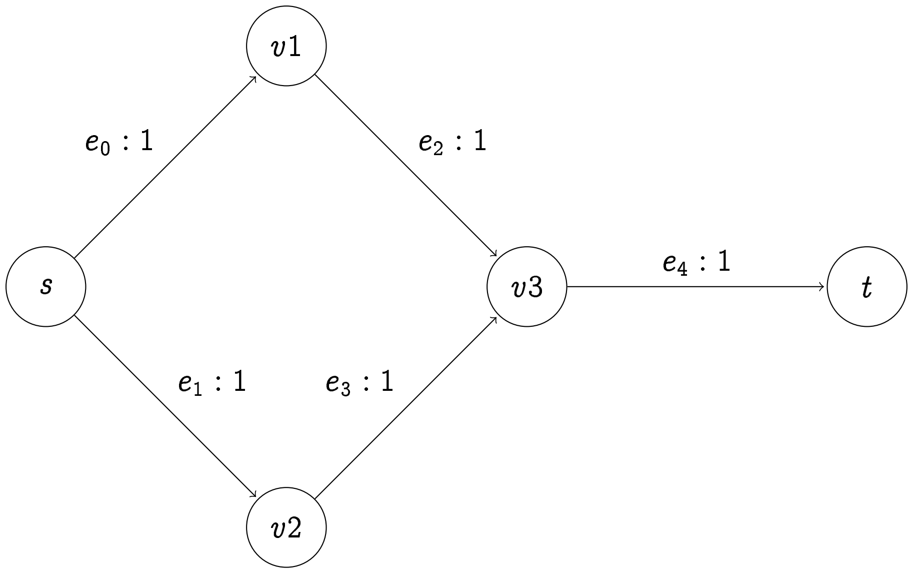
```
    return M[j]

optimalSubset(P = {p_0, p_1, ..., p_n}):
    Allocate empty array M and I each with length n + 1
    run minError(n)
    optSet = [p_n ]
    i = n
    while i > 0:
        optSet.append(p_{I[i]})
        i = I[i]
    return optSet
```

Time complexity: $O(n^2)$
Since computing each M[i] requires a for loop that iterates at most n times (from 0 to n - 1), and inside each for loop the time complexity is O(1), therefore computing M[i] requires O(n). Since the length of M is n + 1 and we compute each M[i] at most once, therefore the algorithm takes $O(n^2)$

Space complexity: $O(n)$
Since we only allocated 2 arrays M and I of length n + 1, therefore space complexity is O(n).

Step Five:
We would compute the quantities in the ascending order. In other words, we could compute opt(0) and fill in M[0] and I[0] first, then compute opt(1) and fill in M[1] and I[1], ..., and at last opt(n) and fill in M[n] and I[n]. This order works because while calculating opt(j) for any j ($0 \leq j \leq n$) we need to know the value of opt(i) such that $0 \leq i < j$. And in the ascending order they are all calculated and stored in M[i] and the time complexity for accessing them is O(1).

Q3
a)
This does not always happen. Example:



As shown by the graph above, every edge of N has capacity 1. Suppose the maximum flow f is pushed through path e_0, e_2, and e_4. In this case, maximum flow is 1 and $f(e_0) = c(e_0) = 1$. Now we decrease $c(e_0)$ by 1. The max flow is still 1, but instead is pushed through path e_1, e_3, and e_4.

b)

maxFlow($N = (V, E), f$):
  // Construct the residual network
  for each edge $e_i = (u_i, v_i) \in$ E:
    if $f(e_i) = c(e_i)$:
      switch e_i from (u, v) to (v, u) and $c(e_i) = f(e_i)$
    if $f(e_i) < c(e_i)$:
      add edge $e_i' = (v, u)$ to E and $c(e_i') = f(e_i)$
      change $c(e_i)$ to $c(e_i) - f(e_i)$
  run breadth first search to look for a s-t path
  if a s-t path p exists:
    update f to get f' such that

$$f'(e) = \begin{cases} f(e) + 1 & \text{if } e \in \{\text{edges on path } p\} \\ f(e) & \text{otherise} \end{cases}$$

    return f'
  return f

determineNewMaxFlow($N = (V, E), f, e_0 = (u_0, v_0)$)

run breadth first search from s to u_0 through edges with none 0 flow to find a $s - u_0$ path p_1

run breadth first search from v_0 to t through edges with none 0 flow to find a $v_0 - t$ path p_2

// To make sure f' is a valid flow when applied to N'

update f to get f' such that

$$f'(e) = \begin{cases} f(e) - 1 & \text{if } e \in \{\text{edges on path } p_1, e_0, \text{edges on path } p_2\} \\ f(e) & \text{otherise} \end{cases}$$

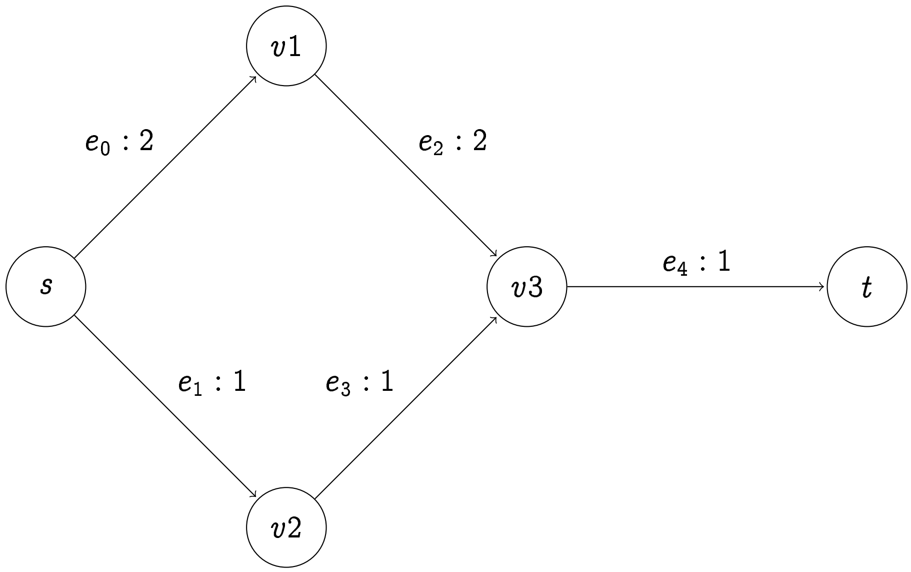
construct $N' = N$ and have $c'(e_0) = c(e_0) - 1$

return maxFlow($N' = (V', E'), f'$)

Justification that the algorithm is correct:
The algorithm determineNewMaxFlow() first finds a s-u-v-t path with edges with none zero flow and decrease the flow by 1 so that after decreasing $c(e_0)$ by 1 the network flow is still valid. Then the algorithm calls maxFlow() which first iterates through all edges of N' to construct the residual network. Then maxFlow() looks for a s-t path in the residual network. If such path does not exist, then it means the current flow is the max flow and the value is $|f|$ - 1; however if a s-t path is found, then we can push 1 unit of flow (since it is not possible to increase the max flow by decreasing $c(e_0)$ by 1), resulting in max flow f.

Worst case runtime:
determineNewMaxFlow($N = (V, E), f, e_0 = (u_0, v_0)$) takes O(V + E):
Runs breadth first search (O(V + E) according to CLRS page 597) twice to find a desired $s - u_0 - v_0 - t$ path, which takes O(V + E). Iterates through E to update f to f'. Iterating through $N = (V, E)$ to construct N' which takes O(V + E).
maxFlow($N = (V, E), f$) takes O(V + E):
Iterates through E' to construct the residual graph which takes O(E). Runs breadth first search to look for a s-t path takes O(V + E). If such s-t path is found updating f to f' requires iterating though at most all the edges which takes O(E)
Therefore the algorithm takes O(V + E)

c)
This does not always happen. Example:



As shown by the graph above, edge e_0 and e_2 has capacity 2 while all the other edges has capacity 1. Suppose the maximum flow f is pushed through path e_0, e_2, and e_4. In this case, $f = 1$ and $f(e_0) = 1$. Now we increase $c(e_0)$ by 1. The max flow f still has value 1, whether or not it is still pushed through e_0, e_2, and e_4 or instead it is pushed through e_1, e_3, and e_4.

d)
maxFlow($N = (V, E), f$):
  // Construct the residual network
  for each edge $e_i = (u_i, v_i) \in$ E:
    if $f(e_i) = c(e_i)$:
      switch e_i from (u, v) to (v, u) and $c(e_i) = f(e_i)$
    if $f(e_i) < c(e_i)$:
      add edge $e_i' = (v, u)$ to E and $c(e_i') = f(e_i)$
      change $c(e_i)$ to $c(e_i) - f(e_i)$
  run breadth first search to look for a s-t path
  if a s-t path p exists:
    update f to get f' such that

$$f'(e) = \begin{cases} f(e) + 1 & \text{if } e \in \{\text{edges on path } p\} \\ f(e) & \text{otherise} \end{cases}$$

    return f'

```
    return f

determineNewMaxFlow(N = (V, E), f, e_0 = (u_0, v_0))
    construct N' = N and have c'(e_0) = c(e_0) + 1
    return maxFlow(N' = (V', E'), f)
```

Justification that the algorithm is correct:
The algorithm determineNewMaxFlow(), after constructing N' , calls maxFlow() which first iterates through all edges of N' to construct the residual network. Then maxFlow() looks for a s-t path in the residual network. If such path does not exist, then it means the current flow f is the max flow; however if a s-t path is found, then we can push 1 unit of flow (since it is not possible to decrease the max flow by increasing $c(e_0)$ by 1), resulting in max flow f' with value $|f| + 1$.

Worst case runtime:
determineNewMaxFlow($N = (V, E), f, e_0 = (u_0, v_0)$) takes O(V + E):
Iterating through $N = (V, E)$ to construct N' takes O(V + E).
maxFlow($N = (V, E), f$) takes O(V + E):
Iterates through E' to construct the residual graph which takes O(E). Runs breadth first search to look for a s-t path takes O(V + E). If such s-t path is found, updating f to f' requires iterating though at most all the edges which takes O(E)
Therefore the algorithm takes O(V + E)

Q4
a)

```
test({r_1, r_2..., r_n}, k):
    U, V, E = {}, {}, {}
    for i = 1 to n:
        create two nodes u_i, v_i (u_i represents stop s(r_i), v_i represents stop e(r_i))
        add u_i to U
        add v_i to V
        create edge (u_i, v_i) with lower bound and capacity 1 then add this edge to E
    for i = 1 to n:
      for j = 1 to n:
        if i! = j:
          if d(r_i) + t(u_i, v_i) + t(v_i, u_j) <= d(r_j):
            create edge (v_i, u_j) with capacity 1 and add it to E.
```

create source node s with supply -k
create sink node t with demand k
for node i in U:
  add edge (s, u_i) with capacity 1 to E
for node j in V:
  add edge (v_j, t) with capacity 1 to E for each node in V
$G' = transfer(G = (U \cup V \cup \{s, t\}, E))$
return $has\_circulation(G')$

(transfer is the way to transfer graph with lower bound to without lower bound in slides 5 page 45, $has\_circulation$ is the claim in lecture slide 5 page 41)

b)
number of edges = O(n)
number of vertices = O(n)
sum of capacities of edges from source vertex = O(n)
Thus, Ford Fulkerson algorithm costs $O(n^3)$ to get maximum flow and other comparisons and trannsfer cost constant time.
So $total = O(n^3)$

c)
Instead of return $has\_circulation(G')$, return the maximum flow number using Ford Fulkerson algorithm for G', which is the minimum number of buses needed.
Time complexity is $O(n^3)$ which is explained in part b.