Question 1

Part a)

Let $x, y \in \{A, C, G, T\}^*$.

For notation, let $x = x_1x_2...x_m$ and $y = y_1y_2...y_n$, where each $x_i, y_j \in \{A, C, G, T\}$ for $i \in \{1, ..., m\}$ and $j \in \{1, ..., n\}$.

youWe will define a table C with $(m + 1) \times (n + 1)$ entries.

Let $C[i, j]$ be the value of the optimal (maximum) score of an alignment between the first i elements of x and the first j elements of y where $i \in \{0, 1, ..., m\}$ and $j \in \{0, 1, ..., n\}$.

The desired solution we will compute would then be $C[m, n]$ where m and n are the lengths of x and y respectively. i.e. $C[m, n]$ is the value of the optimal (maximum) score of an alignment of the first $m = len(x)$ elements of x and the first $n = len(y)$ elements of y.

We will then introduce a second table B in part d) that will help determine the actual alignment using C.

Part b)

Consider the following Bellman equation. Let δ be the scoring matrix. We will then justify its correctness.

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ and } j = 0 \\ C[i, j-1] + \delta(-, y_j) & \text{if } i = 0 \text{ and } j > 0 \\ C[i-1, j] + \delta(x_i, -) & \text{if } i > 0 \text{ and } j = 0 \\ \max \begin{cases} C[i-1, j-1] + \delta(x_i, y_j) \\ C[i-1, j] + \delta(x_i, -) \\ C[i, j-1] + \delta(-, y_j) \end{cases} & \text{if } i > 0 \text{ and } j > 0 \end{cases}$$

Note, the above typesetting of the maximum function in the fourth case was typed that way because the Bellman equation would not have fit horizontally on the page otherwise.

Justification for Correctness

We will consider each of the 4 cases of the Bellman Equation above.

Case 1: For $i = 0$ and $j = 0$, we are considering the first 0 elements of x and the first 0 elements of y. Hence, we get a maximum score of 0 as there are no elements to compare similarity. Hence, $C[i, j] = 0$ is correct.

Case 2: For $i = 0$ and $j > 0$, we are considering the first 0 elements of x and the first $j > 0$ elements of y. Hence, y_j must be aligned with a gap. Hence, the max score can be broken down into $C[i, j-1]$ which is the max score of the first $i = 0$ elements of x with the first $j-1$ elements of y, plus $\delta(-, y_j)$. Hence, $C[i, j] = C[i, j-1] + \delta(-, y_j)$ is correct.

Case 3: For $i > 0$ and $j = 0$, we are considering the first $i > 0$ elements of x and the first $j = 0$ elements of y. Hence, x_i must be aligned with a gap. Hence, the max score can be broken down into $C[i-1, j]$ which is the max score of the first $i-1$ elements of x with the first $j = 0$ elements of y, plus $\delta(x_i, -)$. Hence, $C[i, j] = C[i-1, j] + \delta(x_i, -)$ is correct.

Case 4: For $i > 0$ and $j > 0$, we have three sub-cases to consider. Either we can align x_i with y_j, or we can align x_i with a gap, or we can align y_j with a gap.

Subcase 4 a) If we align x_i with y_j, then the max score we could achieve would be $C[i-1, j-1] + \delta(x_i, y_j)$ where $C[i-1, j-1]$ is the max score from aligning the first $i-1$ elements of x and the first $j-1$ elements of y.

Subcase 4 b) If we align x_i with a gap, then we still must align y_j with something. Hence, the max score we could achieve would be $C[i-1, j] + \delta(x_i, -)$ where $C[i-1, j]$ is the max score from aligning the first $i-1$ elements of x and the first j elements of y since we have already aligned x_i, but have not aligned y_j.

Subcase 4 c) If we align y_j with a gap, then we still must align x_i with something. Hence, the max score we could achieve would be $C[i, j-1] + \delta(-, y_j)$ where $C[i, j-1]$ is the max score from aligning the first i elements of x and the first $j-1$ elements of y since we have already aligned y_j, but have not aligned x_i.

To get the optimal solution, we must take the maximum of the three subcases above.

i.e. The optimal solution is the following for $i > 0$ and $j > 0$.

$$C[i, j] = \max \begin{cases} C[i-1, j-1] + \delta(x_i, y_j) \\ C[i-1, j] + \delta(x_i, -) \\ C[i, j-1] + \delta(-, y_j) \end{cases}$$

Finally, since each non-base case of the Bellman equation involves either the i term or j term getting smaller, we do not have a cyclic definition.

This completes our justification of our Bellman equation.

Note on Bellman Equation: There are likely alternative Bellman equations with slight differences. However, we chose ours because it leads to a natural ordering for a bottom-up implementation for our actual algorithm in parts c) and d).

Please see the next page.

Part c)

Consider the following algorithm which takes input $x, y \in \{A, C, G, T\}^*$ where $x = x_1x_2...x_m$ and $y = y_1y_2...y_n$, where each $x_i, y_j \in \{A, C, G, T\}$ for $i \in \{1, ..., m\}$ and $j \in \{1, ..., n\}$.

MaxAlignmentScore(x, y)

```
1:  $m \leftarrow len(x)$    ▷ Precondition: Assume that $len(x) \leq len(y)$; if not reverse order of $x, y$
2:  $n \leftarrow len(y)$
3:
4:  Let $C$ be a new table with $m+1$ rows and $n+1$ columns. Each entry of $C$ is $C[i,j]$
    where $i \in \{0, 1, ..., m\}$ and $j \in \{0, 1, ..., n\}$.
5:
6:  $C[0,0] \leftarrow 0$                                    ▷ Clause 1 of Bellman Equation
7:
8:  for each $j$ from 1 to $n$ do                           ▷ Clause 2 of Bellman Equation
9:      $C[0,j] \leftarrow C[0,j-1] + \delta(-, y_j)$
10: end for
11:
12: for each $i$ from 1 to $m$ do                           ▷ Clause 3 of Bellman Equation
13:     $C[i,0] \leftarrow C[i-1,0] + \delta(x_i, -)$
14: end for
15:
16: for each $i$ from 1 to $m$ do                           ▷ Clause 4 of Bellman Equation
17:     for each $j$ from 1 to $n$ do
18:         If $j > 2$, delete column $j-2$ to free up memory.
19:         $p \leftarrow C[i-1,j-1] + \delta(x_i, y_j)$
20:         $q \leftarrow C[i-1,j] + \delta(x_i, -)$
21:         $r \leftarrow C[i,j-1] + \delta(-, y_j)$
22:         $C[i,j] \leftarrow \max(p, q, r)$
23:     end for
24: end for
25:
26: return $C[m,n]$
```

Runtime and Space Complexity: Each computation of $C[i,j]$ for any $i \in \{0, 1, ..., m\}$ and $j \in \{0, 1, ..., n\}$ takes $\mathcal{O}(1)$ time. And there are $(m+1) \times (n+1)$ many entries in our table C (if we include deleted columns/entries). i.e. We have $\mathcal{O}(mn)$ many entries in our table that were ever computed where each entry takes $\mathcal{O}(1)$ time to compute. Hence, the worst case running time of our algorithm is $\mathcal{O}(mn)$.

Since we only need to store at most 2 columns, we have that the space complexity becomes $\mathcal{O}(\min(m, n))$. Note, we take the minimum because the roles of x and y are symmetric in our algorithm and we can reverse the roles if it is not the case that $len(x) \leq len(y)$. Please see the next page.

Part d)

We will modify the algorithm from part c) to include a second table B. We will explain table B on the next page after stating the modified algorithm below.

MaxAlignmentScoreModified(x, y)

```
m ← length(x)
n ← length(y)

Let B and C be 2 new tables with m + 1 rows and n + 1 columns each. Each entry of B
and C is B[i, j] and C[i, j] where i ∈ {0, 1, ..., m} and j ∈ {0, 1, ..., n}.

C[0, 0] ← 0                                          ▷ Clause 1 of Bellman Equation
B[0, 0] ← None

for each j from 1 to n do                            ▷ Clause 2 of Bellman Equation
    C[0, j] ← C[0, j − 1] + δ(−, y_j)
    B[0, j] ← "x_gap"
end for

for each i from 1 to m do                            ▷ Clause 3 of Bellman Equation
    C[i, 0] ← C[i − 1, 0] + δ(x_i, −)
    B[i, 0] ← "y_gap"
end for

for each i from 1 to m do                            ▷ Clause 4 of Bellman Equation
    for each j from 1 to n do
        p ← C[i − 1, j − 1] + δ(x_i, y_j)
        q ← C[i − 1, j] + δ(x_i, −)
        r ← C[i, j − 1] + δ(−, y_j)
        s ← max(p, q, r)
        C[i, j] ← s

        if s = p then
            B[i, j] ← "aligned"
        else if s = q then
            B[i, j] ← "y_gap"
        else if s = r then
            B[i, j] ← "x_gap"
        end if
    end for
end for

return B and C                                       ▷ We return both tables now
```

We have modified the algorithm from part c) to include a second table B such that each entry $B[i, j]$ where $i \in \{0, 1, ..., m\}$ and $j \in \{0, 1, ..., n\}$ is such that $B[i, j]$ is equal to either "*aligned*" if x_i and x_j are aligned together, or "*y_gap*" if we align x_i with $-$, or "*x_gap*" if we align $-$ with y_j.

Now consider the following algorithm that will print the actual alignment.

Formatting of Printing: We will describe the formatting of the printing. Consider the example given on the homework handout.

$$- AT - GCC$$

$$TA - CGCA$$

We will instead print out our alignment column-by-column as follows.

$(-, T)$
(A, A)
$(T, -)$
$(-, C)$
(G, G)
(C, C)
(C, A)

The following print algorithm takes as input the table B, the strings x and y, as well as indices $i \in \{0, 1, ..., m\}$ and $j \in \{0, 1, ..., n\}$.

Note, the initial call will be $PrintAlignment(B, x, y, m, n)$ where $m = len(x)$ and $n = len(y)$.

PrintAlignment(B, x, y, i, j)

```
if $i = 0$ and $j = 0$ then
    Do not print anything.
end if

if $B[i, j] =$ "aligned" then
    $PrintAlignment(B, x, y, i - 1, j - 1)$
    print $(x_i, y_j)$
else if $B[i, j] =$ "y_gap" then
    $PrintAlignment(B, x, y, i - 1, j)$
    print $(x_i, -)$
else if $B[i, j] =$ "x_gap" then
    $PrintAlignment(B, x, y, i, j - 1)$
    print $(-, y_j)$
end if
```

Re-analyze time and space complexities:

Now we will re-analyze the time and space complexity of our modified algorithm and the corresponding printing algorithm.

First we will analyze the time and space complexity of MaxAlignmentScoreModified. Each computation of $C[i, j]$ for any $i \in \{0, 1, ..., m\}$ and $j \in \{0, 1, ..., n\}$ takes $\mathcal{O}(1)$ time. And there are $(m+1) \times (n+1)$ many entries in our table C. i.e. We have $\mathcal{O}(mn)$ many entries in our table where each entry takes $\mathcal{O}(1)$ time to compute. The exact same reasoning applies to the table B.

Hence, the overall worst-case runtime of MaxAlignmentScoreModified is $\mathcal{O}(mn)$.

Since we only need to store $\mathcal{O}(mn)$ entries for B and $\mathcal{O}(mn)$ entries for C, we have that our overall space complexity for MaxAlignmentScoreModified is also still $\mathcal{O}(mn)$.

Looking at PrintAlignment, the non-recursive parts take $\mathcal{O}(1)$ time. And each recursive call decrements input i or input j by 1. Hence, the runtime of PrintAlignment is $\mathcal{O}(m + n)$.

So we take $\mathcal{O}(mn)$ time for MaxAlignmentScoreModified and $\mathcal{O}(m + n)$ time for PrintAlignment in the worst case.

If we were to run both procedures, one after the other, the combined worst case runtime of doing both procedures is $\mathcal{O}(mn)$.

Summary: The runtime is $\mathcal{O}(mn)$ and the space complexity is $\mathcal{O}(mn)$ from the two procedures combined, one after the other.

Note: On Piazza, the instructor indicated that we can have a separate print algorithm.

Question 2

Part a)

Let $l_1, l_2, ..., l_n$ and L be our input where $l_i \leq L$ for each $i \in \{1, ..., n\}$.

In any valid division of plank lengths into groups, let d be the largest sum of plank lengths amongst any group in this division.

Let $C[i, j]$ be the minimum possible d in all possible valid divisions restricted to $\{l_1, ..., l_i\}$ where we have at most j groups in any division, and $i, j \in \{1, ..., n\}$.

Note, minimizing the total length of the largest group in all possible divisions allows us to equalize the group lengths as much as possible.

The desired solution to our original problem is then $C[n, n]$ where n is the number of plank lengths in our input, and where we have at most n groups.

Part b)

We will define a Bellman equation for $C[i, j]$ and then explain it more thoroughly in our justification later.

Let s be an array with $n + 1$ entries. Let $s[0] = 0$ and let $s[i] = s[i - 1] + l_i$ for $i \in \{1, ..., n\}$.

In effect, we have that $s[i] = \sum_{t=1}^{i} l_t$ for $i \in \{1, ..., n\}$, and $s[0] = 0$.

Consider the following array α. This will help enforce the fact that each group has length at most L and will help simplify our Bellman equation for $C[i, j]$. We will only use entries of α where $a \geq b$ in our Bellman equation for $C[i, j]$ where $a, b \in \{0, 1, ..., n\}$.

$$\alpha[a, b] = \begin{cases} s[a] - s[b] = \sum_{t=b+1}^{a} l_t & \text{if } s[a] - s[b] \leq L \\ \infty & \text{otherwise} \end{cases}$$

Consider the following Bellman equation.

$$C[i, j] = \begin{cases} s[1] & \text{if } i = 1 \text{ and } j \geq 1 \\ \alpha[i, 0] & \text{if } i > 1 \text{ and } j = 1 \\ \min\limits_{k: 1 \leq k \leq i} \max(C[k, j - 1], \alpha[i, k]) & \text{if } i > 1 \text{ and } j > 1 \end{cases}$$

Note: The above Bellman equation is written using α and s to help with readability. Otherwise we would have more clauses with ∞ in our Bellman equation for $C[i, j]$ to enforce the fact that each group must have total length at most L.

Justification of Bellman Equation:

Clause 1: When $i = 1$ and $j \geq 1$, our problem is restricted to $\{l_1\}$ with at most $j \geq 1$ many groups. In this case we just have one possible division with one group $\{l_1\}$ with length $s[1] = l_1 \leq L$. Hence, the minimum length of the largest group amongst all possible divisions is $s[1]$. Hence, this clause is justified.

Clause 2: When $i > 1$ and $j = 1$, our problem is restricted to $\{l_1, ..., l_i\}$ and at most 1 group. Hence, we have one possible division with one group $\{l_1, ..., l_i\}$. Hence, the length of this group is $\sum_{t=1}^{i} l_t = s[i] = s[i] - s[0]$. But we could have $s[i] - s[0] > L$. In this case we should output ∞, as no actual valid division would exist here. And this is precisely what $\alpha[i, 0]$ does in its piece-wise definition. Hence, this clause is justified.

Clause 3: When $i > 1$ and $j > 1$, we have that for some k such that $1 \leq k \leq i$, the minimum length of the largest group amongst divisions from the first k board lengths with at most $j-1$ groups is $C[k, j-1]$. And then the length of the last group is $\sum_{t=k+1}^{i} l_t = s[i] - s[k] = \alpha[i, k]$ (Note, we could have $\alpha[i, k] = \infty$ if $s[i] - s[k] > L$). We take the maximum of these two values to get the overall length of the maximum group. And we minimize across all k such that $1 \leq k \leq i$ to find the minimum possible length of the maximum group across all divisions. Hence, this clause is justified.

Part c)

Consider the following bottom-up implementation. Please see the next page.

```
**BoardCuts**($l_1, ..., l_n, L$)
Let $s$ be a new table with $n+1$ entries $s[i]$ for $i \in \{0, 1, ..., n\}$.
$s[0] \leftarrow 0$
for $i$ from 1 to $n$ do
    $s[i] \leftarrow s[i-1] + l_i$
end for

Let $\alpha$ be a new table with $(n+1) \times (n+1)$ entries $\alpha[a,b]$ where $a, b \in \{0, 1, ..., n\}$.
for $a$ from 0 to $n$ do
    for $b$ from 0 to $n$ do
        if $s[a] - s[b] \leq L$ then        ▷ Only entries of $\alpha$ where $a \geq b$ are used later
            $\alpha[a,b] \leftarrow s[a] - s[b]$
        else
            $\alpha[a,b] \leftarrow \infty$
        end if
    end for
end for

Let $C$ be a new table with $n \times n$ entries $C[i,j]$ where $i, j \in \{1, ..., n\}$.
for $j$ from 1 to $n$ do                    ▷ Clause 1 of Bellman Equation
    $C[1,j] \leftarrow s[1]$
end for

for $i$ from 2 to $n$ do                    ▷ Clause 2 of Bellman Equation
    $C[i,1] \leftarrow \alpha[i,0]$
end for

for $i$ from 2 to $n$ do                    ▷ Clause 3 of Bellman Equation
    for $j$ from 2 to $n$ do
        $C[i,j] \leftarrow \infty$
        for $k$ from 1 to $i$ do
            if $C[i,j] > max(C[k,j-1], \alpha[i,k])$ then
                $C[i,j] \leftarrow max(C[k,j-1], \alpha[i,k])$
            end if
        end for
    end for
end for

return $C[n,n]$
```

Analysis of Runtime and Space Complexity:

Computing all entries of s takes $\mathcal{O}(n)$ time as there are $\mathcal{O}(n)$ entries where each entry takes constant time to compute.

Computing all entries of α takes $\mathcal{O}(n^2)$ time as there are $\mathcal{O}(n^2)$ entries where each entry takes constant time to compute.

Computing all entries of C takes $\mathcal{O}(n^3)$ time as there are $\mathcal{O}(n^2)$ entries where each entry takes $\mathcal{O}(n)$ time to compute given the minimization of k.

Hence, the total runtime is $\mathcal{O}(n^3)$.

We require $\mathcal{O}(n)$ space for s, and $\mathcal{O}(n^2)$ for α, and $\mathcal{O}(n^2)$ space for C. Hence, the total space complexity is $\mathcal{O}(n^2)$.

Summary: The runtime is $\mathcal{O}(n^3)$ and the space complexity is $\mathcal{O}(n^2)$.

Part d)

Now we will modify our algorithm to actually output the required division. We will separate our algorithm into two functions. And we will introduce a new array B where $B[i, j]$ gives the index of the board length which separates the last group and second-to-last group in our division restricted to the first i boards and where we have at most j groups. i.e. We store the k value in clause 3 of our Bellman equation.

Note, not all entries of B need to be filled when looking at the base cases of our Bellman equation.

Note that by definition, we know that any valid input has an optimal division given that each $l_i \leq L$.

Please see the next page.

BoardCutsModified($l_1, ..., l_n, L$)

```
Let $s$ be a new table with $n+1$ entries $s[i]$ for $i \in \{0, 1, ..., n\}$.
$s[0] \leftarrow 0$
for $i$ from 1 to $n$ do
    $s[i] \leftarrow s[i-1] + l_i$
end for

Let $\alpha$ be a new table with $(n+1) \times (n+1)$ entries $\alpha[a, b]$ where $a, b \in \{0, 1, ..., n\}$.
for $a$ from 0 to $n$ do
    for $b$ from 0 to $n$ do
        if $s[a] - s[b] \leq L$ then          ▷ Only entries of $\alpha$ where $a \geq b$ are used later
            $\alpha[a, b] \leftarrow s[a] - s[b]$
        else
            $\alpha[a, b] \leftarrow \infty$
        end if
    end for
end for

Let $C$ be a new table with $n \times n$ entries $C[i, j]$ where $i, j \in \{1, ..., n\}$.
Let $B$ be a new table with $n \times n$ entries $B[i, j]$ where $i, j \in \{1, ..., n\}$.

for $j$ from 1 to $n$ do                      ▷ Clause 1 of Bellman Equation
    $C[1, j] \leftarrow s[1]$
end for

for $i$ from 2 to $n$ do                      ▷ Clause 2 of Bellman Equation
    $C[i, 1] \leftarrow \alpha[i, 0]$
end for

for $i$ from 2 to $n$ do                      ▷ Clause 3 of Bellman Equation
    for $j$ from 2 to $n$ do
        $C[i, j] \leftarrow \infty$
        for $k$ from 1 to $i$ do
            if $C[i, j] > max(C[k, j-1], \alpha[i, k])$ then
                $C[i, j] \leftarrow max(C[k, j-1], \alpha[i, k])$
                $B[i, j] \leftarrow k$
            end if
        end for
    end for
end for

return $B$ and $C$                            ▷ Note, not all of $B$ will be filled
```

The following division procedure will take table B as input from **BoardCutsModified**, as well as indices i and j and $l_1, ..., l_n$. The initial call will be to $Division(B, n, n, l_1, ..., l_n)$. And note that we know a valid division exists since each $l_i \leq L$.

Division($B, i, j, l_1, ..., l_n$)

```
1:  if $i = 1$ and $j \geq 1$ then
2:      return $\{l_1\}$
3:  end if
4:
5:  if $i > 1$ and $j = 1$ then
6:      return $\{l_1, ..., l_i\}$          ▷ We know a division exists and initial call is $i = n, j = n$
7:  end if
8:
9:  if $i > 1$ and $j > 1$ then
10:     $k \leftarrow B[i, j]$
11:     return $Division(B, k, j-1, l_1, ..., l_n)$ and $\{l_{k+1}, ..., l_i\}$          ▷ Ignore any empty sets
12: end if
```

Analysis of Runtime and Space Complexity:

The runtime of **BoardCutsModified** is exactly the same as **BoardCuts** since the only modification we made was adding the array B which does not asymptotically change the runtime as its entries are computed simultaneously with C. Similarly, the space complexity of **BoardCutsModified** is exactly the same as **BoardCutsModified** as we are simply adding an additional $\mathcal{O}(n^2)$ space complexity from B which does not change the asymptotic result.

Hence, the runtime of **BoardCutsModified** is still $\mathcal{O}(n^3)$, and the space complexity is still $\mathcal{O}(n^2)$.

Looking at the procedure **Division**, we know that calling $Division(B, n, n, l_1, ..., l_n)$ takes $\mathcal{O}(n^2)$ time since each recursive call makes j go down by 1 and possibly makes i go down by 1 or more. Hence, we have at most $\mathcal{O}(n^2)$ recursive calls where each call takes constant time.

Hence, combining the two procedures one after the other, we get a runtime of $\mathcal{O}(n^3)$, and a space complexity of $\mathcal{O}(n^2)$.

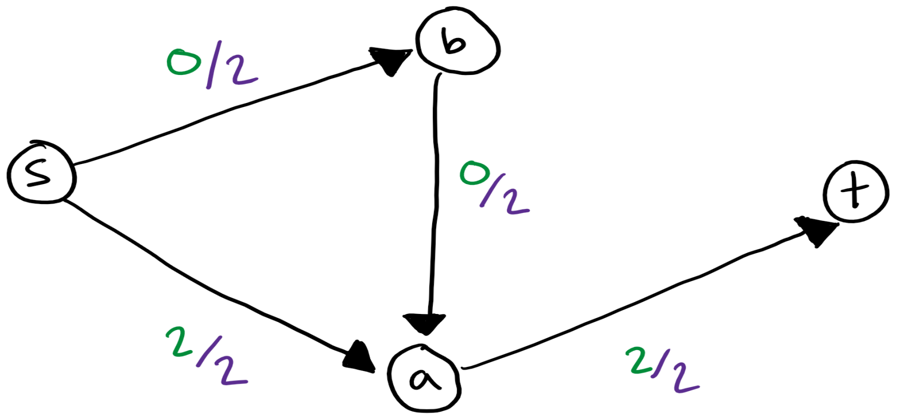
Question 3

Part a)

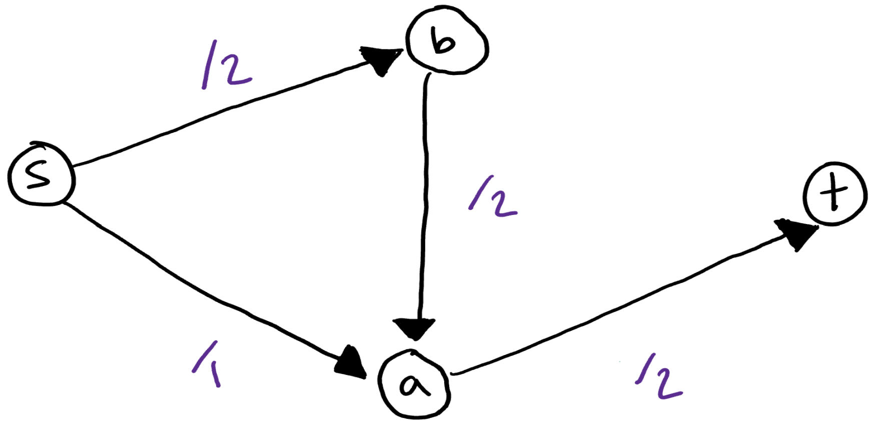
Statement: If $N = (V, E)$ is a network, f^* is a maximum flow in N, $e_0 \in E$ is an edge with $f^*(e_0) = c(e_0)$, and N' is the same network as N except that $c'(e_0) = c(e_0) - 1$, then the maximum flow f' in N' satisfies $|f'| < |f^*|$.

We will disprove this statement by providing a counter-example.

Consider the following network N drawn below with a max flow f^* given on the graph edge labels. Note, the purple denominators are the edge capacities, and the green numerators are the flow. The max flow value is clearly 2 in this simple network. Let $e_0 = (s, a)$. Note, $f^*(e_0) = c(e_0) = 2$.

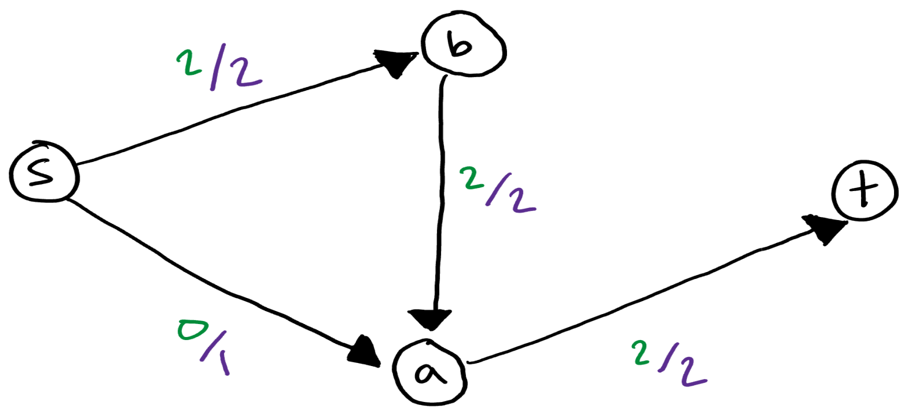


Now let N' be the exact same network as N but where $c'(e_0) = c(e_0) - 1$ when $e_0 = (s, a)$. It is drawn below.



Please turn to the next page.

However, note that N' has a max flow f' which is indicated below with max flow value of 2.



Therefore, $|f'| = |f^*| = 2$. Therefore, the statement is indeed false and disproven.

Part b)

Consider the following algorithm. Note, we will mutate some of our inputs.

ReducedMaxFlow($N = (V, E), f^*, e_0$)

```
1: Denote the edge $e_0$ by $e_0 = (a, b)$.
2: Use breadth-first-search to find a path $P_1$ from source node $s$ to node $a$ in $N$.
3: Use breadth-first-search to find a path $P_2$ from node $b$ to target node $t$ in $N$.
4: Let $P$ be the path $P_1 \cup P_2$.                    ▷ We assume paths are sets of edges
5: Modify $N$ so that $c(e_0) = c(e_0) - 1$.
6: Rename $N$ to $N'$.
7:
8: for each edge $e$ on path $P$ do
9:     $f^*(e) \leftarrow f^*(e) - 1$
10: end for
11:
12: Construct the residual graph corresponding to $N'$ with the modified flow $f^*$.
13: if there is an $s - t$ path in the residual graph corresponding to $N'$ then
14:     Execute one iteration of the Ford-Fulkerson algorithm, updating $f^*$ as necessary.
15: end if
16: Rename $f^*$ to $f'$.
17: return $f'$
```

Brief Argument to Correctness of ReducedMaxFlow:

We will give a brief argument. Let $N = (V, E), f^*, e_0 = (a, b)$ be our input. We assume that $f^*(e_0) = c(e_0)$. Our algorithm uses breadth-first search to find a path P_1 from s to a and a path P_2 from b to t. We then combine P_1 and P_2 into path P.

Path P is an $s-t$ path that includes e_0. Since we reduce $c(e_0)$ by 1 and $f^*(e_0) = c(e_0)$, we must reduce the flow f^* by 1 on each edge $e \in P$ to maintain conservation of flow and satisfy our capacity constraints. And our algorithm does $f^*(e) \leftarrow f^*(e) - 1$ for each $e \in P$. Now f^* satisfies flow conservation and our capacity constraints.

But f^* is not necessarily a max flow in our modified network which we renamed N'. We have that $|f^*|$ is at most 1 less than the max flow value in N'. So if there is an $s-t$ path in the residual graph of N', we execute one iteration of Ford-Fulkerson to guarantee that f^* is indeed a max flow in N'. And we rename f^* to f' and return f'. Therefore, our algorithm is correct.

Analysis of Time Complexity:

Let $n = |V|$ and $m = |E|$. We have 2 breadth-first-searches in lines 2 and 3. This takes $\mathcal{O}(m+n) + \mathcal{O}(m+n) = \mathcal{O}(m+n)$ time.

The for loop on line 8 takes $\mathcal{O}(m)$ time since we have at most $\mathcal{O}(m)$ iterations of the for loop as there are $\mathcal{O}(m)$ edges, and each iteration of the for loop takes constant time.

Line 12 takes $\mathcal{O}(m+n)$ time, and lines 13-15 takes $\mathcal{O}(m+n)$ time as we know from lecture.

Hence, combining all these steps we have that our algorithm takes $\mathcal{O}(m+n)$ time, as required.

Part c)

Consider the following algorithm which takes network $N = (V, E)$ as input and outputs a list of all edges $e_1, ..., e_k$ with the property that if the capacity of any edge in that list is reduced by one unit, the value of the maximum flow in N is also reduced.

```
FlowReducer($N = (V, E)$)
1:  Run the Edmonds-Karp algorithm on network $N$ to find maximal flow. Save the residual
    graph $G_f$ that we get at the end of the last iteration of Edmonds-Karp.
2:
3:  $A^* \leftarrow$ the set of vertices reachable from source node $s$ in $G_f$.
4:  $B^* \leftarrow V \setminus A^*$
5:
6:  $L \leftarrow [\ ]$
7:  for each edge $e = (a, b) \in E$ do
8:      if $a \in A^*$ and $b \in B^*$ then
9:          $L.append(e)$
10:     end if
11: end for
12:
13: return $L$
```

Question 4

Part a)

Note: We will model our solution similar to the solution to Question 3 of Tutorial 4 about professors and courses.

First we will define a flow network $\mathcal{N} = (V, E)$ as follows:

- A set of vertices $V = \{s, t\} \cup \{R_i : 1 \leq i \leq n\} \cup \{C_j : 1 \leq j \leq m\}$. Note that we use capital R_i and capital C_j to distinguish these vertex labels from the given non-negative numbers r_i and c_j.
- A set E with the following edges and capacities.
- An edge $(s, R_i) \in E$ with capacity r_i for each $i \in \{1, ..., n\}$
- An edge $(C_j, t) \in E$ with capacity c_j for each $j \in \{1, ..., m\}$
- An edge $(R_i, C_j) \in E$ with capacity $b_{i,j}$ for each $i \in \{1, ..., n\}$ and $j \in \{1, ..., m\}$

Consider the following algorithm.

MatrixPuzzleSolver

```
1: Compute a maximal flow $f$ on the flow network $\mathcal{N}$ via the Edmonds-Karp algorithm.
   Note that $f$ will be an integral flow since all edge capacities of $\mathcal{N}$ are integers.
2:
3: if $f(s, R_i) = r_i$ for each $i \in \{1, ..., n\}$ and $f(C_j, t) = c_j$ for each $j \in \{1, ..., m\}$ then
4:     A feasible matrix exists.
5:     Let $a_{i,j} \leftarrow f(R_i, C_j)$ for each $i \in \{1, ..., n\}$ and $j \in \{1, ..., m\}$
6:     return the collection of numbers $a_{i,j}$ where $i \in \{1, ..., n\}$ and $j \in \{1, ..., m\}$
7: else
8:     return NIL                                              ▷ No such matrix exists
9: end if
```

Please see the next page.

Part b)

We will prove the correctness of our algorithm.

Def: A maximal flow f is "saturated" if $f(s, R_i) = r_i$ for each $i \in \{1, ..., n\}$ and $f(C_j, t) = c_j$ for each $j \in \{1, ..., m\}$.

We will show that there is a 1-1 correspondence between saturated flows f and valid matrices with entries $a_{i,j}$ satisfying the 3 conditions listed in the problem statement.

Saturated Flow $\Rightarrow$ Valid Matrix: Let f be a saturated flow. Hence, we know that $f(s, R_i) = r_i$ for each $i \in \{1, ..., n\}$ and $f(C_j, t) = c_j$ for each $j \in \{1, ..., m\}$.

Now construct the matrix where each entry $a_{i,j}$ is such that $a_{i,j} = f(R_i, C_j)$.

Since $0 \leq f(R_i, C_j) \leq b_{i,j}$ from our capacity constraints and since $a_{i,j} = f(R_i, C_j)$, we know that $0 \leq a_{i,j} \leq b_{i,j}$ and hence the first condition of a valid matrix solution is satisfied.

Since $f(s, R_i) = r_i$, we know that r_i amount of flow that entered R_i will cumulatively be distributed amongst edges (R_i, C_k) for $k \in \{1, ..., m\}$ by conservation of flow. Hence, we know $r_i = \sum_{k=1}^{m} f(R_i, C_k) = \sum_{k=1}^{m} a_{i,k}$. Hence, our second condition of a valid matrix solution is satisfied.

Finally, since $f(C_j, t) = c_j$, we know that c_j amount of flow leaving C_j will cumulatively have entered C_j from edges (R_k, C_j) for $k \in \{1, ..., n\}$ by conservation of flow. Hence, we know $c_i = \sum_{k=1}^{n} f(R_k, C_j) = \sum_{k=1}^{n} a_{k,j}$. Hence, the third condition of a valid matrix solution is satisfied.

Hence, we have a valid matrix with entries $a_{i,j}$ satisfying our 3 conditions.

Valid Matrix $\Rightarrow$ Saturated Flow: Consider a valid matrix solution with entries $a_{i,j}$ that satisfy the 3 conditions given in the handout. i.e.

1. $0 \leq a_{i,j} \leq b_{i,j}$
2. $r_i = \sum_{k=1}^{m} a_{i,k}$
3. $c_j = \sum_{k=1}^{n} a_{k,j}$

Consider a flow f defined as follows. Let $f(R_i, C_j) = a_{i,j}$ for each $i \in \{1, ..., n\}$ and $j \in \{1, ..., m\}$. Since $0 \leq a_{i,j} \leq b_{i,j}$, this satisfies our capacity constraints in the network $\mathcal{N}$. We will define the rest of f next.

Note, $r_i = \sum_{k=1}^{m} a_{i,k} = \sum_{k=1}^{m} f(R_i, C_k)$. Hence, r_i amount of flow leaves vertex R_i and is dispersed among edges (R_i, C_k) where $k \in \{1, ..., m\}$. By conservation of flow, we know that r_i amount of flow must enter R_i. Hence, we must define $f(s, R_i) = r_i$.

Note, $c_j = \sum_{k=1}^{n} a_{k,j} = \sum_{k=1}^{n} f(R_k, C_j)$. Hence, c_j amount of flow enters vertex C_j cumulatively from the edges (R_k, C_j) where $k \in \{1, ..., n\}$. By conservation of flow, we know that c_j amount of flow must leave C_j. Hence, we must define $f(C_j, t) = c_j$.

So f is now a valid flow for network $\mathcal{N}$ that satisfies conservation of flow and our capacity constraints. And we have that $f(s, R_i) = r_i$ for each $i \in \{1, ..., n\}$ and $f(C_j, t) = c_j$ for each $j \in \{1, ..., m\}$. Hence, f is trivially a max flow, and f is saturated.

So we have shown a 1-1 correspondence between saturated flows f and valid matrix solutions with entries $a_{i,j}$ satisfying our 3 conditions in the problem statement.

Hence, our algorithm which checks the existence of a saturated integral flow and converts it into a valid matrix solution using the above correspondence is indeed correct, as required.

Part c)

Runtime Analysis of MatrixPuzzleSolver:

We know that our network $\mathcal{N}$ has $|V| = 2 + m + n = \mathcal{O}(m + n)$ many vertices and $|E| = m + n + mn = \mathcal{O}(mn)$ many edges.

Hence, step 1 which calculates maximal flow using the Edmonds-Karp algorithm takes $\mathcal{O}(|V||E|^2) = \mathcal{O}((m + n)(mn)^2) = \mathcal{O}(m^3n^2 + m^2n^3)$ time.

And Steps 3-9 takes $\mathcal{O}(mn)$ time since it is dominated by the assignment of each $a_{i,j}$ where $i \in \{1, ..., n\}$ and $j \in \{1, ..., m\}$.

Hence, the overall runtime of **MatrixPuzzleSolver** is $\mathcal{O}(m^3n^2 + m^2n^3)$.