Assignment 3

Question 1

(a) Least Total Cost Recurrence

The goal here is to write down a recurrence for $L[\langle i,j\rangle]$, where $L[\langle i,j\rangle]$ represents the least total cost of going from the source square to square $\langle i,j\rangle$ in the grid. Since this problem possesses optimal substructure, then this means that an optimal solution to a problem contains within it optimal solutions to subproblems. In this case, for the least total cost to get from the source square to any square $\langle i,j\rangle$, there exists within the grid the least total costs to get from the source square to any other square not $\langle i,j\rangle$, so long as it is reachable from the source square and is not an illegal move. This implies that the least total costs are dependent on other least total costs, a property it possesses via optimal substructure.

For any square $\langle i,j\rangle$, let $\langle i',j'\rangle$ represent the square immediately before $\langle i,j\rangle$, where its least total cost is dependent on it. Then, let $l(i',j')$ represent the cost to get from square $\langle i',j'\rangle$ to square $\langle i,j\rangle$ as stored in the cost function $c\colon \{1,n\}^4 \to \mathbb{Z}_0^+ \cup \{\infty\}$. Thus, the following recurrence can be written for $L[\langle i,j\rangle]$ as follows:

$$L[\langle i,j\rangle] = \begin{cases} 0, & if\ \langle i,j\rangle\ is\ the\ source\ square \\ \infty, & if\ i = 1\ and\ \langle i,j\rangle\ is\ not\ the\ source\ square \\ \min\limits_{\substack{j' \geq j-1 \\ j' \leq j+1 \\ 1 \leq j' \leq n}} \{L[\langle i,j\rangle], L[\langle i',j'\rangle] + l(i',j')\}, & otherwise \end{cases}$$

For the third case, this recurrence says that: for any square $\langle i,j\rangle$ that does not fall under the first two cases, take the minimum between either what is already there as the least total cost for square $\langle i,j\rangle$, or take the least total cost for square $\langle i',j'\rangle$ plus the cost to get from square $\langle i',j'\rangle$ to square $\langle i,j\rangle$. For part (c) of this question, this conditional logic is coded in the Python program as follows:

$$if\ (dp[xNext][yNext] > dp[xCurr][yCurr] + cost)$$

This says that if the stored least total cost in square $\langle i,j\rangle$ is greater than the stored least total cost in square $\langle i',j'\rangle$ plus the cost to get from $\langle i',j'\rangle$ to $\langle i,j\rangle$, then update the least total cost of square $\langle i,j\rangle$. This logic is needed because for a square $\langle i,j\rangle$, assuming it is not on the edge of the grid, there are at most 4 ways for its least total cost to be updated: $\langle i-1,j-1\rangle + cost$, $\langle i-1,j\rangle + cost$, $\langle i-1,j+1\rangle + cost$, or no update. This recurrence stores the least of these, assuming validity, as $\langle i',j'\rangle$ and performs the stated comparison.

Assignment 3

(b) Least Cost Path Recurrence

The goal here is to write down a recurrence for $T[\langle i,j\rangle]$, where $T[\langle i,j\rangle]$ represents the least cost path to get from the source square to some square $\langle i,j\rangle$. Similar to what was mentioned in part (a), the optimal or least cost path can be "lopped off" or "eaten into" from the right side of the problem, i.e., starting from the destination and going backwards towards the source. First, going in the forward direction of the algorithm, the final least total cost will be obtained once it processes all the set of moves in the cost function c. The total cost in the set of destination squares $\langle n,\cdot\rangle$ with the lowest value is the last value in the optimal path, since the algorithm seeks the least cost path.

From here, the recurrence can be realized by working backwards, meaning from an optimal destination square $\langle i,j\rangle$, determine all the immediately previous squares that it could have come from. If at the left edge of the grid $\langle i,1\rangle$, it would have either come from $\langle i-1,1\rangle$ or $\langle i-1,2\rangle$. Similarly, if at the right edge of the grid $\langle i,n\rangle$, it would have either come from $\langle i-1,n\rangle$ or $\langle i-1,n-1\rangle$. Lastly, if anywhere in between $\langle i,j\rangle$ (not including source squares), it would have come from either $\langle i-1,j-1\rangle$, $\langle i-1,j\rangle$, or $\langle i-1,j+1\rangle$.

The algorithm determines which one it comes from by choosing the preceding square with the least total cost from the source up to that point, hence it is dependent on the $L[\langle i,j\rangle]$ function from part (a). However, for the algorithm this is already known, as the program in part (c) tracks not only the optimal squares to take, but also the parent of those optimal squares, or in other words, where they came from when building the dynamic programming array. This strategy can be extended for every square in the optimal path, in a backwards manner, until the source square is reached. With that being said, the following recurrence can be written for $T[\langle i,j\rangle]$ as follows:

$$T[\langle i,j\rangle] = \begin{cases} NIL, & if\ L[\langle i,j\rangle] = 0\ or\ \infty \\ T[\langle i-1,j-1\rangle], & if\ L[\langle i,j\rangle] = L[\langle i-1,j-1\rangle] \\ T[\langle i-1,j\rangle], & if\ L[\langle i,j\rangle] = L[\langle i-1,j\rangle] \\ T[\langle i-1,j+1\rangle], & otherwise \end{cases}$$

In words, for some square $\langle i,j\rangle$, the algorithm chooses the least cost square from one of $\langle i-1,j-1\rangle$, $\langle i-1,j\rangle$, and $\langle i-1,j+1\rangle$, and appends that to its least cost path. This decision is based on each of their costs $L[\cdot]$ from part (a). If $\langle i',j'\rangle$ represents the least cost immediately previous square, then this is taken because $L[\langle i',j'\rangle]$ is the least total cost.

Paolo Torres

Assignment 3

Question 2

(a) **Optimal Substructure**

Optimal substructure is the property that an optimal solution to a problem contains within it optimal solutions to subproblems. We are given as input a set of stops $\{s_1, \ldots, s_n\}$, where each s_{i+1} is reachable from s_i if started at the beginning of the day, and $s_{i+1} > s_i$ in terms of distance from the source A. For single-source minimal stops: if $s_0 \rightsquigarrow s_i \rightsquigarrow s_n$ is a minimal stop path from s_0 to s_n, then the $s_0 \rightsquigarrow s_i$ sub-path is a minimal stop path from s_0 to s_i. Suppose $o_{i_1}, o_{i_2}, \ldots, o_{i_k}$ is an optimal sequence of stops that get us from A to B such that the distance from source A is maximized for each stop. A sub-problem for which $o_{i_1}, \ldots, o_{i_{k-1}}$ has to be an optimal solution: suppose in the input, the stops $o_1, \ldots, o_n$ are ordered by increasing distance from A. Then: $o_i, \ldots, o_{i_{k-1}}$ has to be an optimal solution to all stops that end at or before s_{i_k}. In other words, if B is stop s_{n+1}, the optimal solution is some $o_{i_1}, o_{i_2}, \ldots, o_{i_k}$. Then, if B becomes stop s_n, then the optimal solution becomes $o_{i_1}, o_{i_2}, \ldots, o_{i_{k-1}}$. Thus, no matter what stop B is at, the optimal solution stays the same up to that stop B. This is the optimal substructure that this problem possesses.

(b) Feasibility

The goal is to prove that an instance $s_0, \ldots, s_{n+1}$ of the problem is feasible if and only if for every consecutive pair $\langle s_i, s_{i+1} \rangle$ in the input, we can drive from s_i to s_{i+1} within a day.

Proof by contradiction: suppose the drive cannot be made. Since the drive could not be made, then we must have stopped at s_i and never have been able to make it to s_{i+1}. Let d be the maximum distance we can drive within a day. Since we could not make it to stop s_{i+1}, then $s_i + d < s_{i+1}$, i.e., the distance between the stops is greater than what we can possibly drive. Since the input claims there is some path from s_0 to s_{n+1}, i.e., is feasible, then there must exist some sub-path in that path that starts before s_{i+1} and ends at or after s_{i+1}. Since the drive could not have been made from s_i, then it must have been made from some stop s_k where $s_k < s_i$. But then we have $s_k + d < s_i + d < s_{i+1}$, so this path from s_k to s_{i+1} is also not possible. We have reached our desired contradiction, in that the assumption that the drive cannot be made is wrong, therefore for every consecutive pair $\langle s_i, s_{i+1} \rangle$ in the input, we can in fact drive from s_i to s_{i+1} within a day.

Assignment 3

(c) Fewest Stops Recurrence

The goal is to write a recurrence for $m[j]$, where $m[j]$ is the fewest stops we have to make to get from s_0 to s_j. Let j be the stop number, which will allow our recurrence to represent the stops we must make. For every stop, we can choose to either keep going if we can make it to the next stop before nighttime or stop if we cannot make it. In this case, we want to choose the option which minimizes our number of stops. Therefore, the recurrence is:

$$m[j] = \begin{cases} 0, & if\ s = t \\ m[j+1], & if\ s \to t \\ 1 + m[j+1], & if\ s \nrightarrow t \\ \infty, & otherwise \end{cases}$$

In words: if we have reached stop t from stop s, we do not need any more stops so return 0. If we can get from stop s to stop t ($s \to t$), then simply proceed to the next stop without counting a stop. Conversely, if we cannot get from stop s to stop t ($s \nrightarrow t$), then add 1 to the stop count and proceed to the next stop. Lastly, if the input is infeasible, return mnemonic ∞.

(d) Pseudocode

```
const int NumMinStops(const vector<int> stops) {

   const int numStops = stops.size();

   vector<int> m(numStops + 1, 0);

   for (int j = 0; j < numStops; j++) {

      for (int i = j; i >= 0; i--) {

         if (stops[i] → stops[j]) {

            m[j + 1] = m[j];

         } else if (stops[i] ↛ stops[j]) {

            m[j + 1] = 1 + m[j];

         }
```

```
        }
    }
    if (m.back() != 0) {
        return m.back();
    }
    return ∞;
}
```

The algorithm was written out fully in C++ to be able to test with different inputs for the set of stops, namely, $s_0, \ldots, s_{n+1}$. Afterwards, the following notations were used: $\rightarrow$ to represent being able to get from s to t, $\nrightarrow$ to represent not being able to get from s and t, and ∞ to represent there being no feasible sequence.

The function starts out by initializing the dynamic programming array (m in this case), of the size of the number of stops, all set to a value of 0. Then, the algorithm loops through each stop j and does the following: for each stop i up to stop j, update the dynamic programming array. If the stop can be made ($stops[i] \rightarrow stops[j]$), then make no changes to the count in that part of the array ($m[j+1] = m[j]$). Conversely, if the stop cannot be made ($stops[i] \nrightarrow stops[j]$), then add 1 to its stop count before proceeding to the next one ($m[j+1] = 1 + m[j]$). Lastly, if stop j has been reached ($stops[i] = stops[j]$), then that value is automatically 0 since it was initialized to 0 in the first place. Finally, the last element in the array is the answer: if it has been updated then return that result, else return the mnemonic ∞, as this means that no path from A to B has been found.

Let n be the number of acceptable stops in the input $s_0, \ldots, s_{n+1}$. The worst-case time efficiency of this algorithm is $\Theta(n^2)$. First, we must iterate through each of the n stops once, which results in a time complexity of exactly $\Theta(n)$. Then, for each of the n stops, the algorithm must loop from 0 to $n-1$ to update the dynamic programming array to track the number of stops. This operation also results in a time complexity of exactly $\Theta(n)$. Thus, in the worst-case, the algorithm loops through n stops, and for each of them, loops n number of times to update the array. Therefore, the time efficiency of this algorithm is $\Theta(n^2)$.

The space efficiency of this algorithm is $\Theta(n)$. The data structure used to store the minimum number of stops is a vector (array) of size $numStops + 1$, where $numStops$ is the size of the input $s_0, \dots, s_{n+1}$. And since n is the number of acceptable stops in the input $s_0, \dots, s_{n+1}$, then $numStops = n$. Also note that the $+1$ constant can be omitted from the space efficiency. Therefore, the space efficiency of this algorithm is $\Theta(n)$.

Assignment 3

Question 3

(a) Lower and Upper Bounds on Alignment Length

The goal here is to determine the lower- and upper-bounds on the alignment length of the edit distance problem. To help answer this question, code was written in C++ using dynamic programming to test with different inputs to see the results. The following snippet of the code contains the most important part of the program, which is the build-up of the dynamic programming array as follows:

```
for (int i = 1; i <= n; i++) {
    dp[i][0] = i;
}
for (int j = 1; j <= m; j++) {
    dp[0][j] = j;
}
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        if (x[i-1] == y[j-1]) {
            dp[i][j] = dp[i-1][j-1];
        } else {
            dp[i][j] = min(dp[i-1][j-1],
                           min(dp[i-1][j], dp[i][j-1])) + 1;
        }
    }
```

Paolo Torres

```
}
```

Just like from Lecture 7c, using $x = SNOWY$ and $y = SUNNY$, we get an alignment length of 3 with this algorithm. Furthermore, this algorithm follows exactly the recurrence realized for edit distance from the lecture:

$$E(i,j) = \begin{cases} i, & if\ j = 0 \\ j, & if\ i = 0 \\ \min\{1 + E(i-1,j), 1 + E(i,j-1), E(i-1,j-1)\}, & if\ i > 0, j > 0\ and\ x_i = y_j \\ \min\{1 + E(i-1,j), 1 + E(i,j-1), 1 + E(i-1,j-1)\}, & otherwise \end{cases}$$

This algorithm follows a bottom-up approach, where it fills in a table $dp[0 \dots n, 0 \dots m]$, starting at $[1,1]$ and ending at $[n,m]$, and then returns $dp[n,m]$ as the answer. The algorithm contains four loops: the first two correspond to the first two cases in the recurrence above. For the nested loop, it iterates $n \times m$ times, and for each letter: if they are equal, i.e., $x[i-1] == y[j-1]$, then leave the character unchanged for a cost of 0. If they are not equal, then take the minimum between substitution ($dp[i-1][i-1]$), deletion ($dp[i-1][j]$), and insertion ($dp[i][j-1]$), all for a cost of 1.

Now, analyzing time efficiency, the costliest operation in the algorithm is the nested for loop that iterates $n \times m$ times in order to update the dynamic programming array. Thus, this has a time efficiency of $O(nm)$. In addition, the first two loops run in times $O(n)$ and $O(m)$, respectively. And assuming the worst-case scenario, which corresponds to deleting every character in x and inserting every character in y, the total upper bound time efficiency is $O(n + m + nm) = O(nm)$. Since the nested loop must iterate through the entirety of the dynamic programming array to analyze the strings, then this also results in a lower bound time efficiency of $\Omega(nm)$. For lower bound, since all characters from string x must appear in sequence in the top row, and similarly for string y in the bottom row, then it makes sense that the algorithm would still have to iterate through the entire array to analyze the whole string. Therefore, this also results in a tight bound of $\Theta(nm)$ for the edit distance problem.

(b) Worst-Case Number of Possible Alignments

The goal here is to prove, assuming $n = m$, that the number of possible alignments between the lower- and upper-bounds of part (a) is $\Omega(2^n)$. If we look at the recursive version of the solution, we have the following recurrence:

Assignment 3

$$E(n,m) = \begin{cases} n, & if\ m = 0 \\ m, & if\ n = 0 \\ E(n-1, m-1), & if\ n > 0, m > 0\ and\ x_n = y_m \\ 1 + \min\{E(n-1,m), E(n,m-1), E(n-1,m-1)\}, & otherwise \end{cases}$$

First, in the case that $n \neq m$, we need to consider all three operations: substitution, deletion, and insertion. So, going from right to left, if the letters are the same, then leave the character unchanged for a cost of 0. Else, recurse between the three operations and take the minimum of those three. For each recursive call, in the worst-case, the algorithm would have to once again recurse each of the three options and continue this pattern until the beginning of the string. This leads to a worst-case time efficiency of $\Omega(3^n)$, which happens when the two strings have no letters matching.

Now, take the assumption that $n = m$. Since the strings are now the same length, the algorithm can be simplified in the case of insertion and deletion. If we consider an alignment where the top and bottom rows are each some length l, then we have $l - n$ "_" characters in each of the top and bottom rows. Since we are guaranteed that the strings are of the same length, then insertion and deletion can effectively be collapsed into one operation, simplifying the recursive step.

For example, let $x = a\_bc$ and $y = xy\_z$. For both x and y, in the worst-case, there are each n different ways for the "_" to appear within the strings. This leads to a total of n^2 different ways for the "_" to appear. Next, consider the ways in which the string transformation can happen. Namely, either from $x \to y$ or $y \to x$. If we consider $x \to y$, then only substitution and insertion are required. Similarly, if we consider $y \to x$, then only substitution and deletion are required. This property reveals that, since the string sizes are identical, then the algorithm only needs to consider two of the three operations at a time. Mathematically, either:

$$E(n,m) = \begin{cases} n, & if\ m = 0 \\ m, & if\ n = 0 \\ E(n-1, m-1), & if\ n > 0, m > 0\ and\ x_n = y_m \\ 1 + \min\{E(n-1,m), E(n-1,m-1)\}, & otherwise \end{cases}$$

Or:

$$E(n,m) = \begin{cases} n, & if\ m = 0 \\ m, & if\ n = 0 \\ E(n-1, m-1), & if\ n > 0, m > 0\ and\ x_n = y_m \\ 1 + \min\{E(n,m-1), E(n-1,m-1)\}, & otherwise \end{cases}$$

In both recurrences, there are now only two operations to recurse on, depending on what string transformation the algorithm decides to do. Hence, since we are guaranteed that the string sizes are the same, namely n, then the worst-case time efficiency simplifies to $\Omega(2^n)$.

Paolo Torres