1. **Solution:** Let McKing(i,l) denote the maximum profit that can be achieved if only the locations $L_i, L_{i+1}, ..L_{n-1}, L_n$ were to be considered, and at most l restaurants could be opened. Note that $McKing(1, k)$ is the answer we wish to compute.

   We also append a dummy location L_{n+1}, for reasons explained later, with distance from the starting point $m_{n+1} = \infty$, and profit $p_{n+1} = 0$.

   To solve the problem resursively, we observe that an optimal solution for $McKing(i, l)$ will either have a restaurant opened at location i or not. This leads to the following recurrence:

$$McKing(i,l) = \begin{cases} 0 & i > n \text{ or } l = 0 \\ \max\{p_i + McKing(next(i), l-1), McKing(i+1, l)\} & \text{otherwise} \end{cases}$$

   where $next(i) = \min\{j : m_j \geq m_i + D\}$ is the next available location that is at least distance D away from the current location at L_i. By a simple for loop, all $next(i), \forall i = 1, \cdots, n$ can be calculated in $O(n)$ time. For some i, if every location after i is at most D meters away, we return the dummy location $(n+1)$ as $next(i)$.

   To compute $McKing(1, k)$ efficiently, a naive way to memorize answers to all subproblems is to store all values in a two-dimensional array $McKing[1, \cdots, n+1; 0, \cdots, k]$. However, the space required for such an array is $O(nk)$. We note that a subproblem in the column indexed by l only depends on subproblems in the same column or the previous column indexed by $l-1$. Therefore, only two instead of k columns need to be memorized and the space required reduces to $O(n)$. The following algorithm uses this idea to ensure space efficiency:

```
McKing(L[1..n], D, k):
  for i ← n down to 1
      Vals[i,0] ← 0
      Vals[i,1] ← 0
  col ← 0
  for l ← 0 to k
      for i ← n down to 1
          if i > n or l = 0
              Vals[i,col] ← 0
          else
              Vals[i,col] ← max{p_i + Vals[next(i), 1 − col], Vals[i+1, col]}
      col ← 1 − col
  return Vals[1, 1 − col]
```

   Running time: there are $O(nk)$ distinct subproblems. If $next(i)$ is preprocessed, each subproblem takes $O(1)$ time. There will be $O(n)$ extra space needed to store $next(i), \forall i \in [n]$, and $O(n)$ time to compute these values. The algorithm in all requires $O(n) + O(nk) = O(nk)$ time. ■

Rubric: Standard DP rubric. 10 points total. Additionally:
- **0 point** for solving the problem without using constraint k.
- **5 points max** for no recurrence/pseudo code but correct description and idea presented.
- We do not penalize for not preprocessing $next(i)$.

2. **Solution:** Let $SCSS(i,j,k)$ be the length of the shortest common supersequence of $x_1 \dots x_i$, $y_1 \dots y_j$, and $z_1 \dots z_k$. Also, define x_0, y_0, z_0 as three new symbols not used in X, Y, Z. The recurrence below satisfies this definition.

$$SCSS(i,j,k) = \begin{cases} & \textbf{Base cases:} \\ \infty & \text{if } i < 0 \text{ or } j < 0 \text{ or } k < 0 \\ 0 & \text{if } i = 0,\ j = 0,\ k = 0 \\ & \textbf{When all rightmost characters match:} \\ 1 + SCSS(i-1,\ j-1,\ k-1) & \text{if } x_i = y_j = z_k \\ & \textbf{When two rightmost characters match:} \\ 1 + \min \begin{cases} SCSS(i,\ j-1,\ k-1) \\ SCSS(i-1,\ j,\ k) \end{cases} & \text{if } y_j = z_k \neq x_i \\ 1 + \min \begin{cases} SCSS(i-1,\ j,\ k-1) \\ SCSS(i,\ j-1,\ k) \end{cases} & \text{if } x_i = z_k \neq y_j \\ 1 + \min \begin{cases} SCSS(i-1,\ j-1,\ k) \\ SCSS(i,\ j,\ k-1) \end{cases} & \text{if } x_i = y_j \neq z_k \\ & \textbf{When all rightmost characters are different:} \\ 1 + \min \begin{cases} SCSS(i-1,\ j,\ k) \\ SCSS(i,\ j-1,\ k) \\ SCSS(i,\ j,\ k-1) \end{cases} & \begin{array}{l} \text{(otherwise)} \\ \text{if } x_i \neq y_j,\ y_j \neq z_k,\ x_i \neq z_k \end{array} \end{cases}$$

For a dynamic programming algorithm using this recurrence, we memoize $SCSS(i,j,k)$ in a 3D array with dimensions $(r+1) \times (s+1) \times (t+1)$, noting that there are extra cells for when $i,j,k = 0$. All cells depend on cells with lower coordinates, so we fill the array like so: fill the array in 2D slices from lowest k to highest k, where for each 2D slice, we fill the i dimension in increasing order, and for each i, we fill the j dimension in increasing order. The goal is to compute $SCSS(r,s,t)$, which gives the shortest common subsequence for all of X, Y, and Z.

Using the filling order, each individual array cell can be filled in constant time; filling a single array cell requires at most three constant-time memoized lookups in a min function, which is still constant time overall for each cell. To fill the whole array, $O(r \cdot s \cdot t)$ cells must be filled. This gives $O(rst)$ as the running time and $O(rst)$ as the space.[1] ■

> **Rubric:** Standard DP rubric. Furthermore:
> - **5 points max** for giving super-sequence solution for 2 strings instead of 3.
> - **5 points max** for no recurrence/pseudo code but correct description and idea presented.

3. **Solution:**
   - **Part (a)** Given a rooted tree T and a node v we let #MATCHINGS(v) denote the number of distinct matchings in the subtree T_v of T rooted at v. We develop a recurrence for #MATCHINGS(v) as follows. If v is a leaf, then it is a base case and

[1]You don't need to specify the space requirement. Also, technically, it takes $O(\log(r+s+t))$ bits to write down the value in each cell, since $r+s+t$ is the longest that the shortest common supersequence might be if you concatenate the three sequences, so the space and time bounds might be better stated as $O(rst \log(r+s+t))$.

we have $\#\text{Matchings}(v) = 1$ since $\emptyset$ is the only feasible matching. Otherwise we count the matchings as follows. Let $\mathcal{M}(v)$ be the set of all matchings in T_v. We partition $\mathcal{M}(v)$ into $\mathcal{M}_1(v)$ which consists of all matchings that do not contain any edges incident to v, and $\mathcal{M}_2(v)$ as the set of matchings that contain an edge incident to v. Note that $|\mathcal{M}(v)| = |\mathcal{M}_1(v)| + |\mathcal{M}_2(v)|$. We observe that

$$|\mathcal{M}_1(v)| = \prod_{a \in \text{children}(v)} \#\text{Matchings}(a)$$

since any matching $M \in \mathcal{M}_1(v)$ can be uniquely decomposed into matchings in the subtrees $T_a, a \in \text{children}(v)$. To count $\mathcal{M}_2(v)$ we see that any matching $M \in \mathcal{M}_2(v)$ must have exactly one edge incident to v. For a child u of v let $\mathcal{M}_2(v,u) = \{M \in \mathcal{M}_2(v) \mid M \text{ contains the edge } (v,u)\}$ be the subset of matchings that contain the edge (v,u). If (v,u) is in a matching M then no other edges incident to u can be in M. Via the same reasoning as before we see that

$$|\mathcal{M}_2(v,u)| = \left(\prod_{b \in \text{children}(u)} \#\text{Matchings}(b)\right)\left(\prod_{a \in \text{children}(v), a \neq u} \#\text{Matchings}(a)\right).$$

We can simplify the above formula by noticing that the first term is the same as $|\mathcal{M}_1(u)|$ and the second term also can be simplified. We thus have

$$|\mathcal{M}_2(v,u)| = |\mathcal{M}_1(u)| \times \frac{|\mathcal{M}_1(v)|}{\#\text{Matchings}(u)}.$$

Thus, we obtain the following recursive formula:

$$\#\text{Matchings}(v) = \begin{cases} 1 & \text{if } v \text{ is a leaf} \\ |\mathcal{M}_1(v)| + \sum_{u \in \text{children}(v)} |\mathcal{M}_2(v,u)| & \text{otherwise} \end{cases}$$

The algorithm needs to output $\#\text{Matchings}(r)$ where r is the root of T.

It is convenient to store in two arrays, the quantities, $\#\text{Matchings}(v)$ and $|\mathcal{M}_1(v)|$ for each node v. We need to compute these quantities in a bottom up fashion. For this purpose we compute a post-traversal $v_1, v_2, \ldots, v_n$ of the nodes and evaluate the quantities $\#\text{Matchings}(v_i)$ and $|\mathcal{M}_1(v_i)|$ in this order. We see that the computation at a node v requires $O(k)$ multiplications and $O(k)$ divisions and $O(k)$ additions where k is the degree of the node. Thus the total number of arithmetic operations is $\sum_{v \in V} \deg(v)$ which is $O(n)$. If we want to avoid division we need to do k^2 multiplications at a node with degree k. This results in $O(n^2)$ work using only multiplications and additions. One can be more clever and reduce the number of multiplications but we will not describe those optimizations here.

- **Part (b)**

  Suppose we have a path on n nodes. We root it one of the end of the path which implies that each node has a single child and each subtree is a path. In this case the recurrence can be written simply in terms of n the number of nodes in the path. Let $T(n)$ be number of matching on a path of n nodes when it is rooted at one end. It is convenient to consider the case when $n = 0$ and set $T(0) = 0$. When $n = 1$ we have a single leaf in which case $T(1) = 1$. Apply the recursion in the preceding part and simplifying we obtain the following

$$T(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ T(n-1) + T(n-2) & \text{if } n \geq 2 \end{cases}$$

This recursion should remind you of the Fibonacci recurrence except for the base cases. We note that $T(2) = 1$. Thus we see that $T(n) = Fib(n-1)$ for all $n \geq 1$. One can use the formula for $Fib(n)$ or see that $T(n) \geq 2T(n-2)$ to conclude that $T(n) \geq 2^{\lfloor (n-1)/2 \rfloor}$ for all $n \geq 2$. In particular this implies that $T(n) \geq 2^{249}$ for $n = 500$. But then $T(n)$ would require at least 249 bits which would not fit in a 64 bit word.

- **Part (c)** We notice that, since the answer size can grow exponentially with n we cannot store it necessarily in one word even if that word can fit n. We observe that the number of matchings in a graph on m edges is at most 2^m since there are at most 2^m subsets of edges. In a tree T, $m = n-1$ and hence the maximum number we ever need to store is 2^{n-1}. Thus we need to only store numbers with at most n bits. To implement the algorithm in part (a) without overflow we will explicitly store each of the two arrays as n-bit integers rather than 64-bit words. Various modern object oriented programming languages offer this ability using BigInt class.

  We now analyze the running-time assuming that each intermediate number has at most n bits. Note that the total number of arithmetic operations was $O(n)$ if divisions are allowed. Otherwise we had $O(n^2)$ arithmetic operations if only additions and multiplications are allowed. Let $\alpha(n)$ denote the time to multiply two n-bit integers and let $\beta(n)$ be the time to divide two n bit integers; in general division is harder than multiplication so we will assume that $\beta(n) \geq \alpha(n)$. . We have two potential running times. $O(n^2\alpha(n))$ if we used only multiplications and $O(n\beta(n))$ if we used divisions. Standard multiplication of two n bit integers takes $O(n^2)$ time and hence $\alpha(n) = O(n^2)$. But we say that Karatsuba's algorithm can be used to show that $\alpha(n) = O(n^{\log 1.5})$. We did not see an algorithm for division but it is not hard to convince yourself that division can be reduced to multiplication via binary search. Dividing two n bit numbers (integer division) can be solved via $O(n)$ calls to multiplication of n bit numbers. Thus we have $\beta(n) \leq n\alpha(n)$. Hence we end up, in both methods, with a running time of $O(n^2\alpha(n))$. However, there are more advanced techniques that show that division can essentially be done in roughly the same time as multiplication. Thus, using those methods we can in fact obtain a running time which is $O(n\alpha(n))$.

■

Rubric: Part (a): Standard DP rubric scaled to 5 pts
Part (b): 2 pts. 1pt for deriving the recurrence and 1pt for analyzing the value for $n = 500$
Part (c): 3 pts
1pt for stating and justifying the number of bits required to store the count
1 pt for implementation change from part (a)
1pt for correct runtime and justification.