1. The McKing chain wants to open several restaurants along Red street in Shampoo-Banana. The possible locations are at $L_1, L_2, \ldots, L_n$ where L_i is at distance m_i meters from the start of Red street. Assume that the street is a straight line and the locations are in increasing order of distance from the starting point (thus $0 \leq m_1 < m_2 < \ldots < m_n$). McKing has collected some data indicating that opening a restaurant at location L_i will cost c_i independent of where the other restaurants are located. However, the city of Shampoo-Banana has a zoning law which requires that any two McKing locations should be D or more meters apart. McKing wants to open k restaurants based on their budget and profitablity considerations. Describe an algorithm that McKing can use to figure out the minimum cost it needs to incur to open k restaurants while satisfying the city's zoning law. Your algorithm should output ∞ if k restaurants cannot be opened due to the zoning law. For full credit your algorithm should use only $O(n)$ space; note that you cannot assume that k is a fixed constant.

Solution: Let $McKing(i, \ell)$ denote the minimum cost that can be achieved if only the locations $L_i, L_{i+1}, ..L_{n-1}, L_n$ were to be considered, and ℓ restaurants must be opened. Note that $McKing(1, k)$ is the final answer we wish to compute.

To deal with edge cases, we add a sentinel location L_{n+1} with distance from the starting point $m_{n+1} = \infty$, and cost $c_{n+1} = \infty$.

To solve the problem recursively, 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:

$$McKing(i,\ell) = \begin{cases} \infty & i > n \text{ and } \ell > 0 \\ 0 & \ell = 0 \\ \min \left\{ \begin{array}{c} c_i + McKing(next(i), \ell - 1), \\ McKing(i+1, \ell) \end{array} \right\} & \text{otherwise} \end{cases}$$

where $next(i) = \min \left\{ j \;\middle|\; m_j \geq m_i + D \right\}$ is the next available location that is at least distance D away from the current location at L_i. The first condition in the base case represents the situation where we have not placed all the required restaurants, and the second is where no more restaurants must be placed.

We will first precompute $next(i)$ for $1 \leq i \leq n$. Observe that for $i < j$, $next(i) \leq next(j)$, so the following algorithm correctly computes $next(i)$ for all i:

```
PrecomputeNext(L[1..n], D)
  j ← 1
  for i ← 1 to n
      while j ≤ n and L[j] < L[i] + D
          j ← j + 1
      next[i] ← j
```

Despite the nested loops, this algorithm actually only takes $O(n)$ time: the variable j can only be incremented n times throughout the entirety of the algorithm.

We can naïvely memoize into a two-dimensional array $McKing[1..n+1, 0..k]$. The subproblem $McKing[i,\ell]$ depends on subproblems in column $\ell-1$ and column ℓ, and rows j where $j > i$. We thus memoize by filling column by column from left to right in the outer loop, and row by row from bottom to top in the inner loop.

```
MinCostMcKing(L[1..n], D, k):
  PrecomputeNext(L[1..n], D)
  for ℓ ← 0 to k
      for i ← n+1 down to 1
          if i > n and ℓ > 0
              McKing[i, ℓ] ← ∞
          else if ℓ = 0
              McKing[i, ℓ] ← 0
          else
              McKing[i, ℓ] ← min{c_i + McKing[next[i], ℓ − 1], McKing[i + 1, ℓ]}
  return McKing[1, k]
```

This scheme requires $O(nk)$ space. To reduce the space needed, we will use the fact that $McKing[i,\ell]$ depends on subproblems in ***only*** columns $\ell-1$ and ℓ. Therefore, instead of keeping all k columns in memory, we only need to keep two: the current column being computed and the most recently computed column. This reduces the space required to $\boldsymbol{O(n)}$.

```
SpaceEfficientMinCostMcKing(L[1..n], D, k):
  PrecomputeNext(L[1..n], D)
  col ← 0
  for ℓ ← 0 to k
      for i ← n+1 down to 1
          if i > n and ℓ > 0
              McKing[i, col] ← ∞
          else if ℓ = 0
              McKing[i, col] ← 0
          else
              McKing[i, col] ← min{c_i + McKing[next[i], 1 − col], McKing[i + 1, col]}
      col ← 1 − col
  return McKing[1, 1 − col]
```

There are $O(nk)$ distinct subproblems. Since $next(i)$ is precomputed for each i, each subproblem takes $O(1)$ time. Since preprocessing $next(i)$ for $1 \le i \le n$ takes $O(n)$ time, overall the algorithm runs in $\boldsymbol{O(nk)}$ ***time***. ■

Rubric: 10 points: standard DP rubric.

- **0 point** for solving the problem without using constraint k.
- **5 points max** for no recurrence/pseudo code but correct description and idea presented.
- **8 points max** for $\omega(n)$ space.
- No penalty for slower runtime due to not precomputing $next(i)$.

2. Let $X = x_1, x_2, \ldots, x_r$, $Y = y_1, y_2, \ldots, y_s$ and $Z = z_1, z_2, \ldots, z_t$ be three sequences. A common *supersequence* of X, Y and Z is another sequence W such that X, Y and Z are subsequences of W. Suppose $X = a, b, d, c$ and $Y = b, a, b, e, d$ and $Z = b, e, d, c$. A simple common supersequence of X, Y and Z is the concatenation of X, Y and Z which is $a, b, d, c, b, a, b, e, d, b, e, d, c$ and has length 13. A shorter one is b, a, b, e, d, c which has length 6. Describe an efficient algorithm to compute the *length* of the shortest common supersequence of three given sequences X, Y and Z. You may want to first solve the two sequence problem to get you started.

Solution: Let $SCSS(i, j, k)$ be the length of the shortest common supersequence (SCSS) of $x_1 \ldots x_i$, $y_1 \ldots y_j$, and $z_1 \ldots z_k$. To easier handle the case where one or more of the three substrings is empty, we will add sentinel values x_0, y_0, z_0 as three new symbols not used in X, Y, Z.

We can observe the following possibilities for $x_1 \ldots x_i, y_1 \ldots y_j, z_1 \ldots z_k$:

- Suppose all rightmost characters match, i.e., $x_i = y_j = z_k$, the length of the SCSS is $SCSS(i-1, j-1, k-1) + 1$. The $+1$ is due to this matching rightmost character.
- Suppose two of the rightmost characters match, i.e., one of the following cases occurs: $x_i = y_j \neq z_k$, $x_i = z_k \neq y_j$, or $y_j = z_k \neq x_i$.
  If $x_i = y_j \neq z_k$, then either:
  - The length of the SCSS is $SCSS(i-1, j-1, k) + 1$, corresponding to the rightmost character of the SCSS being x_i $(= y_j)$, or
  - The length of the SCSS is $SCSS(i, j, k-1) + 1$, corresponding to the rightmost character of the SCSS being z_k.

  The other two cases can be derived in a similar fashion.
- Suppose all three rightmost characters are different, i.e., $x_i \neq y_j$, $y_j \neq z_k$, $x_i \neq z_k$. Then either
  - The length of the SCSS is $SCSS(i-1, j, k)+1$, corresponding to the rightmost character of the SCSS being x_i,
  - The length of the SCSS is $SCSS(i, j-1, k)+1$, corresponding to the rightmost character of the SCSS being y_j, or
  - The length of the SCSS is $SCSS(i, j, k-1)+1$, corresponding to the rightmost character of the SCSS being z_k.
- The case analysis above breaks in the following cases, which become the base cases of our recursion:
  - If $i = j = k = 0$, then all three strings are empty, and so the SCSS is just ε.
  - Any of the above cases may include the possibility that one or two of i, j, k are 0, but not all three, and so may make a recursive call to a case where $i < 0$, $j < 0$, or $k < 0$. To indicate that this should not count as a valid scenario, we will set the length of the SCSS to be ∞.

The preceding explanation is far more detailed than necessary for full credit; we include it to help illuminate the pieces of the recurrence below.

In summary, $SCSS(i,j,k)$ satisfies the following recurrence:

$$SCSS(i,j,k) = \begin{cases} \infty & \text{if } i<0,\ j<0, \text{ or } k<0 \\ 0 & \text{if } i=j=k=0 \\ 1+SCSS(i-1,j-1,k-1) & \text{if } x_i = y_j = z_k \\ 1+\min\left\{\begin{array}{c} SCSS(i,j-1,k-1) \\ SCSS(i-1,j,k) \end{array}\right\} & \text{if } y_j = z_k \neq x_i \\ 1+\min\left\{\begin{array}{c} SCSS(i-1,j,k-1) \\ SCSS(i,j-1,k) \end{array}\right\} & \text{if } x_i = z_k \neq y_j \\ 1+\min\left\{\begin{array}{c} SCSS(i-1,j-1,k) \\ SCSS(i,j,k-1) \end{array}\right\} & \text{if } x_i = y_j \neq z_k \\ 1+\min\left\{\begin{array}{c} SCSS(i-1,j,k) \\ SCSS(i,j-1,k) \\ SCSS(i,j,k-1) \end{array}\right\} & \text{otherwise} \end{cases}$$

For a dynamic programming algorithm using this recurrence, we memoize $SCSS(i,j,k)$ in a three-dimensional array with dimensions $(r+2)\times(s+2)\times(t+2)$, where the extra cells are for when each variable is 0 or -1. Each cell depends only on cells with smaller coordinates, so we fill the array in increasing order for each variable, in any order of the three variables. The goal is to compute $\boldsymbol{SCSS(r,s,t)}$, which gives the length of the shortest common subsequence for all of X, Y, and Z.

For those who prefer it, the iterative pseudocode can be found on the next page (though it is not necessary for full credit).

Each individual array cell can be filled in constant time since 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, $O(rst)$ cells must be filled. This gives $\boldsymbol{O(rst)}$ as the running time and $O(rst)$ as the space. The space requirement is not necessary for full credit. 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 $O(rst\log(r+s+t))$.

```
ShortestCommonSuperseq(X[1..r], Y[1..s], Z[1..t]):
  for j ← 0 to s
      for k ← 0 to t
          SCSS[−1, j, k] ← ∞
  for i ← 0 to r
      for k ← 0 to t
          SCSS[i, −1, k] ← ∞
  for i ← 0 to r
      for j ← 0 to s
          SCSS[i, j, −1] ← ∞
  for i ← 0 to r
      for j ← 0 to s
          for k ← 0 to t
              if i = j = k = 0
                  SCSS[i, j, k] ← 0
              else if x_i = y_j = z_k
                  SCSS[i, j, k] ← 1 + SCSS[i − 1, j − 1, k − 1]
              else if x_i = y_j ≠ z_k
                  SCSS[i, j, k] ← 1 + min{SCSS[i − 1, j − 1, k], SCSS[i, j, k − 1]}
              else if x_i = z_k ≠ y_j
                  SCSS[i, j, k] ← 1 + min{SCSS[i − 1, j, k − 1], SCSS[i, j − 1, k]}
              else if y_j = z_k ≠ x_i
                  SCSS[i, j, k] ← 1 + min{SCSS[i, j − 1, k − 1], SCSS[i − 1, j, k]}
              else
                  SCSS[i, j, k] ← 1 + min{SCSS[i − 1, j, k], SCSS[i, j − 1, k],
                                          SCSS[i, j, k − 1]}
  return SCSS[r, s, t]
```

■

Solution (Incorrect): It is tempting to claim that a SCSS of X, Y, Z can be found by setting A to be a SCSS of X, Y and computing a SCSS of A and Z.

This claim is false. Consider the example $X =$ **ab**, $Y =$ **ac**, $Z =$ **acab**. Then $A =$ **abc** is a SCSS of X and Y in which case **acabc** is a SCSS of A and Z with length five. On the other hand, $Z =$ **acab** is a SCSS of X, Y, Z with length four. ■

Rubric: 10 points: standard DP rubric. Furthermore:

- **5 points max** for computing the shortest common supersequence of *two* strings instead of three.
- **5 points max** for no recurrence/pseudo code but correct description and idea presented.
- **0 points** for claiming that the SCSS of X, Y, Z can be found by setting A to be the SCSS of X, Y and computing the SCSS of A and Z.

3. In lecture we saw an efficient algorithm to compute a maximum weight independent set in a given tree $T = (V, E)$.

   (a) Given a tree $T = (V, E)$ describe an efficient algorithm to *count* the number of distinct independent sets in T. Two independent sets S_1 and S_2 are distinct if they are not identical as sets of vertices. Note that the empty set counts as a valid independent set.

Solution: Given a tree T and a vertex u, let $NumIndSets(u)$ be the number of distinct independent sets in the subtree $T(u)$ rooted at u.

We can divide the distinct independent sets of $T(u)$ into two categories: the independent sets containing u, and the independent sets *not* containing u. We will compute the two counts separately and then add them together.

- First consider the independent sets that do not contain u. Each such independent set decomposes into a choice for each child v of u, of some independent set of $T(v)$. This gives $\prod_{v \text{ child of } u} NumIndSets(v)$ independent sets that do not contain u.
- Now consider the independent sets that do contain u. Each such independent set cannot contain any children of u, and so decomposes into u along with a choice for each *grand*child v of u, of some independent set of $T(v)$. This gives $\prod_{v \text{ grandchild of } u} NumIndSets(v)$ independent sets that do contain u.
- If u is a leaf, the computations above result in empty products, which we interpret as evaluating to 1. Thus no explicit base case is necessary.

In summary:

$$NumIndSets(u) = \prod_{v \text{ child of } u} NumIndSets(v) + \prod_{v \text{ grandchild of } u} NumIndSets(v).$$

To evaluate $NumIndSets(u)$ we need to have computed the values for all children and grandchildren of u, and so we will use a post-order traversal as the memoization order.

```
CountIndSets(T):
  v_1, ..., v_n ← a post-order traversal of the nodes of T
  for i ← 1 to n
      numWithout ← 1
      numWith ← 1
      for each j such that v_j is a child of v_i
          numWithout ← numWithout · NumIndSets[j]
          for each k such that v_k is a child of v_j ⟨⟨i.e., v_k is a grandchild of v_i⟩⟩
              numWith ← numWith · NumIndSets[k]
      NumIndSets[i] ← numWithout + numWith
  return NumIndSets[n] ⟨⟨v_n is the root of T⟩⟩
```

We assume constant-time arithmetic. On the one hand, each multiplication corresponds to one access to $NumIndSets[i]$ for some index i. On the other hand, for each vertex v_i, $NumIndSets[i]$ is only accessed by its parent and grandparent. Thus the total number of accesses and multiplications is $O(n)$. There is one additional addition in the computation for each i, for a total of n additions. In summary, the algorithm takes ***O(n) time***. ■

Solution: Given a tree T and a vertex u, we can divide the distinct independent sets of the subtree $T(u)$ rooted at u into two categories: the independent sets containing u, and the independent sets *not* containing u.

Let $NumIndSets(u, b)$ be the number of distinct independent sets in the subtree $T(u)$ that *do* contain u if $b = 1$, and the number that *do not* contain u if $b = 0$. Then the number of independent sets of $T(u)$ is given by $NumIndSets(u, 1) + NumIndSets(u, 0)$.

- $NumIndSets(u, 1)$ counts the number of independent sets that contain u. Each such independent set decomposes into u along with a choice for each child v of u, of some independent set of $T(v)$ that does not contain v, so

$$NumIndSets(u, 1) = \prod_{v \text{ child of } u} NumIndSets(v, 0).$$

- $NumIndSets(u, 0)$ counts the number of independent sets that do not contain u. Each such independent set decomposes into a choice for each child v of u, of some independent set of $T(v)$. Recalling that the number of independent sets of $T(v)$ is given by $NumIndSets(v, 1) + NumIndSets(v, 0)$,

$$NumIndSets(u, 0) = \prod_{v \text{ child of } u} \big(NumIndSets(v, 1) + NumIndSets(v, 0)\big).$$

- If u is a leaf, the computations above result in empty products, which we interpret as evaluating to 1. Thus no explicit base case is necessary.

To evaluate $NumIndSets(u, b)$ we need to have computed the values for all children u, and so we will use a post-order traversal as the memoization order.

```
CountIndSets(T):
  v_1, ..., v_n ← a post-order traversal of the nodes of T
  for i ← 1 to n
      NumIndSets[i, 1] ← 1
      NumIndSets[i, 0] ← 1
      for each j such that v_j is a child of v_i
          NumIndSets[i, 1] ← NumIndSets[i, 1] · NumIndSets[j, 0]
          NumIndSets[i, 0] ← NumIndSets[i, 0]
                              · (NumIndSets[j, 1] + NumIndSets[j, 0])
  return NumIndSets[n, 1] + NumIndSets[n, 0] ⟨⟨v_n is the root of T⟩⟩
```

We assume constant-time arithmetic. For each vertex u, we perform three array accesses and three arithmetic operations (two multiplications and an addition) for each child of u. Finally, we perform one final addition in the return statement. In total, the algorithm takes $O(1) + \sum_{u \in V} O(\deg(u)) = \boldsymbol{O(n)}$ ***time***. ■

(b) What is the exact number of independent sets if the tree is a star with a center and $n-1$ leaves? Would the answer for a star with $n = 500$ fit in a 64-bit integer word? Briefly justify your answer.

Solution: Interpreting the center of the star as being the root, the analyses in the solutions to part (a) tell us that there is one independent set that contains the center, and 2^{n-1} independent sets that do not contain the center, for a total of $\mathbf{2^{n-1}+1}$ ***independent sets***.

For $n = 500$, this value is $2^{499}+1$, which requires $\lceil \log_2(2^{499}+1) \rceil = 500$ bits to write down. This ***does not fit*** into a 64-bit integer word. ■

(c) How would you implement your counting algorithm from part (a), more carefully, to run on a 64 bit machine? Accounting for this more careful implementation, what is the running time of your algorithm?

Solution: Since the answer size can grow exponentially in n, we will not be able to store the answer in a single fixed-size word. However, each independent set of T is a subset of the vertices of T, so bounding loosely we will never need to write down a number larger than 2^n, which can be written down using n bits. *Thus instead of storing the numbers in 64-bit words, we can store them as n-bit integers via some sort of BigInteger implementation.*

We will thus analyze the running time of the algorithm assuming that each intermediate number consists of at most n bits. Let $\alpha(n)$ be the running time for multiplying two n-digit numbers. As we saw in lecture, naïve implementations for multiplication gives $\alpha(n) = O(n^2)$, Karatsuba's algorithm gives $\alpha(n) = O(n^{\log_2 3})$, and the recent result of Harvey and van der Hoeven puts $\alpha(n)$ at $O(n \log n)$. The standard algorithm for adding two n-bit integers takes $O(n)$ time, which cannot be improved, since it takes $\Omega(n)$ time to write down their sum.

Both solutions given for part (a) involved $O(n)$ multiplications, $O(n)$ additions, and $O(n)$ array accesses, giving a running time of $O(n\alpha(n) + n^2 + n)$, which simplifies to $O(n^3)$ via naïve multiplication, or $\boldsymbol{O(n^2 \log n)}$ via Harvey and van der Hoeven's fast multiplication algorithm. ■

Rubric: 10 points.

(a) 5 points for part (a): standard DP rubric divided by two.

- Max 4 points for slower runtime (when analyzed assuming constant time arithmetic)

(b) 2 points for part (b)

- +1 for computing the number of independent sets for a star with $n-1$ leaves
- +1 for correctly assessing that the number does not fit in a 64-bit word for $n = 500$.

(c) 3 points for part (c)

- +1 for justifying the number of bits needed to store the answer
- +1 for explaining the implementation change (saying "BigInteger" suffices)
- +1 for runtime analysis (no penalty for slower multiplication implementations)

Rubric: Standard dynamic programming rubric
For problems worth 10 points:

- 6 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  - + 1 point for a clear **English** description of the function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.) **Automatic zero if the English description is missing.**
  - + 1 point for stating how to call your function to get the final answer.
  - + 1 point for base case(s). −½ for one *minor* bug, like a typo or an off-by-one error.
  - + 3 points for recursive case(s). −1 for each *minor* bug, like a typo or an off-by-one error. **No credit for the rest of the problem if the recursive case(s) are incorrect.**
- 4 points for details of the dynamic programming algorithm
  - + 1 point for describing the memoization data structure
  - + 2 points for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested loops, be sure to specify the nesting order.
  - + 1 point for time analysis
- It is *not* necessary to state a space bound.
- For problems that ask for an algorithm that computes an optimal *structure*—such as a subset, partition, subsequence, or tree—an algorithm that computes only the *value* or *cost* of the optimal structure is sufficient for full credit, unless the problem says otherwise.
- Official solutions usually include pseudocode for the final iterative dynamic programming algorithm, ***but iterative pseudocode is not required for full credit***. If your solution includes iterative pseudocode, you do not need to separately describe the recurrence, memoization structure, or evaluation order. (But you still need to describe the underlying recursive function in English.)
- Official solutions will provide target time bounds. Algorithms that are faster than this target are worth more points; slower algorithms are worth fewer points, typically by 2 or 3 points (out of 10) for each factor of n. Partial credit is scaled to the new maximum score, and all points above 10 are recorded as extra credit.

  We rarely include these target time bounds in the actual questions, because when we have included them, significantly more students turned in algorithms that meet the target time bound but didn't work (earning 0/10) instead of correct algorithms that are slower than the target time bound (earning 8/10).