Question 1 (Date Planning, 50 points). Debbie is trying to find a spouse. Over the next \( n \) calendar days she has \( m \) suitors to consider. For each suitor, she has the option of dating them for a fixed interval of days (from day \( s_i \) to day \( e_i \)). Each suitor has a known probability \( p_i \) of proving to be a good match for Debbie, and probabilities \( p_{i,d} \) that Debbie will find out on day \( d \) of dating them that they will not be a good match.

Debbie does not want to date more than one suitor at a time and once she starts dating one of them will continue to do so until she determines one way or the other whether or not they are a match.

Provide a polynomial time algorithm to determine the best probability Debbie can obtain of finding a match during this time period.

Solution 1.

Algorithm. We solve this problem via dynamic programming.

Without loss of generality, we will only consider suitors whose \( s_i \) or \( e_i \) fall between 0 and \( n - 1 \), inclusive. We will keep a table \( T \) with \( n + 1 \) indices such that the \( d \)th index will store the best probability for Debbie if she were to start dating on day \( d \). We initialize \( T[n] = 0 \) as Debbie has 0% probability of finding a suitor if she is still single on the \( n + 1 \)th day. Starting from day \( d = n - 1 \) and working backwards until we hit day 0, we check to see if any suitors have a start date on day \( d \). If so we check to see if dating one of them will produce a higher probability of success than \( T[d + 1] \) (the probability of a match if she were to date no one and wait a day). For each newly available suitor \( j \), we compute \( p_{\text{match},j} \) by:

\[
p_j + \sum_{k=s_j}^{e_j} p_{j,k} \times T[k + 1]
\]

In other words, for all days \( k : s_j \leq k \leq e_j \), we sum the probabilities of failing to find a spouse if Debbie started dating \( j \), found out \( j \) was a bad match on day \( k \), and dated optimally on day \( k + 1 \). We add this sum to the probability that \( j \) is a good match in general, giving us the probability of Debbie finding some match if she dates \( j \).

Once we have checked every \( j \), we compute \( \text{best}_j = \max(p_{\text{match},j}) \) over all \( j \) and assign \( T[d] = \max(\text{best}_j, T[d + 1]) \).

At the end of our algorithm, we return \( T[0] \) which will be the best probability Debbie can obtain of finding a match during the next \( n \) calendar days.

Correctness. Let \( T[k] \) be the best probability Debbie has of finding a match if she starts dating on day \( k \). Let \( J \) be the set of suitors that are available to date on day \( k \). For \( k = n \), \( T[k] = 0 \) since Debbie only dates suitors in the first \( n \) days. For \( 0 \leq k < n \) we see that \( T[k] \) must come from Debbie either dating some \( j \in J \) or not dating anyone and waiting a day. Specifically, \( T[k] \) is the maximum probability over these two possibilities. To determine which \( j \) Debbie should date (if any), we compute \( p_{\text{match},j} \): the best probability that Debbie will eventually match with someone if she starts dating suitor \( j \). Notice that

(a). There is a \( p_j \) chance suitor \( j \) is a good match.
(b). Debbie has a \( p_{j,k'} \) chance of finding out \( j \) is not a good match on day \( k' \) and can resume dating on day \( k' + 1 \), for \( s_i \leq k' \leq e_i \) which implies

\[
p_{\text{match},j} = p_j + \sum_{k'=s_j}^{e_j} p_{j,k'} \cdot T[k'+1]
\]

. Then \( T[k] \) is simply the maximum over all \( p_{\text{match},j} \) and \( T[k+1] \).

To prove correctness of \( T[k] \) for all \( 0 \leq k \leq n \), we proceed by induction on \( k \), going backwards from \( n \) to 0. For our base case \( k = n \), we see that Debbie has 0% probability of finding a match since she cannot date past the first \( n \) days, so our base case of \( T[n] = 0 \) is correct. Let’s assume \( T[k] \) is true for all \( k > r \geq 0 \). To show it is true for \( k - 1 \), notice that our inductive hypothesis implies \( T[(k-1)+c] \) is computed correctly for \( 0 < c \leq n - (k-1) \). This implies that \( T[(k-1)+1] \) is computed correctly and, since \( k-1 = s_j \) and \( s_j < e_j < n \), \( p_j + \sum_{k'=s_j}^{e_j} p_{j,k'} \cdot T[k'+1] \), is computed correctly for all \( j \in J \). Since \( T[k-1] \) is simply the maximum of these values, this implies \( T[k-1] \) is computed correctly. This implies \( T[k] \) is correct for \( k \geq 0 \), completing our proof of correctness.

**Complexity.** Our runtime is \( O(mn) \).

For each day \( d \), we iterate over \( m \) suitors to check if \( s_i == d \) which is \( O(mn) \). For each suitor \( j \) that satisfies this constraint, we compute \( p_{\text{match},j} = p_j + \sum_{k'=s_j}^{e_j} p_{j,k'} \cdot T[k+1] \). because we use a table \( T \) to store Debbie’s probability of success in the future and \( T[k+1] \) is computed before \( T[k] \), we can determine the value of \( T[k+1] \) in constant time, giving us the ability to only do one add and \((e_i - s_i) \) multiply/add operations. In the worst case, this is \( O(n) \) operations. We then find the maximum of these values, best. Since suitors can only have a single start date, we only ever consider a suitor once. This implies that we do this \( O(n) \) operations a total of \( m \) times throughout the entire algorithm making this step \( O(mn) \) overall. Since we compare \( best_j \) to \( T[d+1] \) for each day \( d \), this is \( O(n) \) Therefore the total runtime of our algorithm is \( O(mn) + O(mn) + O(n) = O(mn) \) which is polynomial.

**Question 2** (Longest Common Substring on a Budget, 50 points). Give an algorithm that given three strings \( a_1 \), \( b_1 \), \( c_1 \), \( a_2 \), \( b_2 \), \( c_2 \), \( a_n \), \( b_n \), \( c_n \), a cost assigned to each character and a real number \( V \) computes the length of the longest sequence \( x_1 x_2 \ldots x_k \) that is a common subsequence of each of the three sequences and so that the total cost of all the characters in our sequence is at most \( V \). Your algorithm should run in polynomial time. Note that the costs are not necessarily integers, and so algorithms that produce a table with an entry for every possible value of the cost will be too slow.

**Solution 2.** We can solve the problem by Dynamic programming. Our subproblem \( LC(i,j,k,l) \) will be the lowest cost (LC) subsequence that is a common subsequence of \( A_1, \ldots, A_j, B_1 \ldots B_j, C_1 \ldots C_k \) of length exactly \( l \). With this in mind, we can make the following recurrence:

\[
LC(i,j,k,l) = \min(LC(i-1,j,k,l), LC(i,j-1,k,l), LC(i,j,k-1,l), LC(i-1,j-1,k-1,l-1) + \text{cost}(A_i))
\]

where the last option is only available if \( A_i = B_j = C_k \). Our base case is a little complicated. \( LC(i,j,k,0) = 0 \) because all length 0 subsequences are free. Also, if \( l > \min(i,j,k) \) then \( LC(i,j,k,l) = \infty \) because you can’t have a subsequence of a string that is longer than the original string.

With this in mind, we can make an \((n+1) \times (n+1) \times (n+1) \times (n+1) \) table where \( Table[i][j][k][l] \) is the best value of the first \( i,j,k \) characters of \( A,B,C \) with length \( l \). table in increasing \( i,j,k,l \) in that order. (You can imagine the algorithm as \( j \) nested loops where the first loop iterates on \( i \) from 1 to \( n \), the second on \( j \) from 1 to \( n \), etc.) Then at the very end, we look at the \( Table[n][n][n][x] \) values for all \( x \), and find the largest \( x \) such that this entry is less than \( V \), and return that \( x \). This \( x \) will be the length of the longest common subsequence of \( A,B,C \) that doesn’t cost more than \( V \).

**Complexity.** There are \( n^4 \) many subproblems and each subproblem can be solved in a constant number of comparisons, which gives us the overall runtime of \( \Theta(n^4) \), which is clearly polynomial time

**Correctness.** So we can proceed by strong induction on \( i,j,k,l \). We need to prove our base cases are correct. If we want a length 0 subsequence, we can get this trivially by having an empty subsequence and this clearly
costs 0. The other base case is if \( l > \min(i, j, k) \) and this is clearly impossible. Assume without loss of
genitality that \( l > i \). The largest possible subsequence of \( A_i \) has length \( i \), so it is impossible to generate a
subsequence of length greater than \( i \), so this has a best value of \( \infty \). Assume that we have correctly computed
the values of \( LC(i', j', k', l') \) where at least one of \( i', j', k', l' \) is less than \( i, j, k, l \). We just show that we
correctly compute \( LC(i, j, k, l) \). If any of \( i,j,k,l \) are 0, we can use analysis we used above to show that this is
 correct. We also showed that if \( l > \min(i, j, k) \) then the \( LC(i, j, k, l) = \infty \) Ok. Now we know that none of
\( i,j,k,l \) are 0 and \( l \) is at least as large as \( \min(i, j, k) \). Now we want to prove that
\[
LC(i, j, k, l) = \min(LC(i - 1, j, k, l), LC(i, j - 1, k, l), LC(i, j, k - 1, l), LC(i - 1, j - 1, k - 1, l - 1) + \text{cost}(A_i))
\]
where the last option is only available if \( A_i = B_j = C_k \). If we look at \( A_i, B_j, C_k \), we can use a similar argument
for the recurrence of longest common subsequence. For each of \( A_i, B_j, C_k \), we can either include the letter
in our subsequence, or not include the letter in our subsequence. If we include any of the last letters, all
of the last letters can be the same, and the value of this would be \( LC(i - 1, j - 1, k - 1, l - 1) + \text{cost}(A_i) \).
Or, we can take a length \( l \) subsequence where one of the 3 "last" letters are omitted. This corresponds
to \( LC(i - 1, j, k, l), LC(i, j - 1, k, l), LC(i, j, k - 1, l) \). Since we cover all the ways we can have a length \( l \)
subsequence of the first \( i,j,k \) letters of \( A,B,C \) respectively, we can take a minimum over their costs and our
recurrence is correct. This complete the induction. We iterate over Table\([n][n][n][i]\) for all \( i \) and find the
highest length we can afford with our value \( v \) and a linear pass from high \( i \) going down to 0 will take care of
this.

Question 3 (Extra credit, 1 point). Where have I been getting character names for homework problems
from this quarter?

Solution 3. The Big Bang Theory!