1. (a) Describe an algorithm that finds $k$ intervals with minimum total squared length that cover $X$. The running time of your algorithm should be a simple function of $n$ and $k$.

**Solution:** For any non-negative integers $j$ and $\ell$, let $MTSL(i, \ell)$ denote the Minimum Total Squared Length of $\ell$ intervals that cover the prefix $X[1..j]$. We need to compute $MTSL(n, k)$. This function satisfies the following recurrence:

$$MTSL(j, \ell) = \begin{cases} 0 & \text{if } j = 0 \\ \infty & \text{if } \ell = 0 \text{ and } j > 0 \\ \min_{1 \leq i \leq j} \left( MTSL(i - 1, \ell - 1) + (X[j] - X[i])^2 \right) & \text{otherwise} \end{cases}$$

We can cover the empty set with any number of zero-length intervals. Covering a nonempty set requires at least one interval. In the most general setting, we correctly consider every possible rightmost interval $X[i..j]$. (If we define $\min \emptyset = \infty$, the second base case is redundant.)

We can memoize this function into a two-dimensional array $MTSL[0..n, 0..k]$ in either standard row-major order or standard column-major order in $O(n^2 k)$ time:

```
for \ell \leftarrow 0 to k
    MTSL[0, \ell] \leftarrow 0
for j \leftarrow 1 to n
    MTSL[j, \ell] \leftarrow \infty
    for i \leftarrow 1 to j
        MTSL[j, \ell] \leftarrow \min \{MTSL[j, \ell], MTSL(i - 1, \ell - 1) + (X[j] - X[i])^2\}
return MTSL(n, k)
```

**Rubric:** These are not the only correct solutions. Only one correct evaluation order is required for full credit. Explicit iterative pseudocode is not required for full credit; see the instructions for dynamic programming algorithms in HW1.

(b) Consider the two-dimensional matrix $M[1..n, 1..n]$ defined as follows:

$$M[i, j] = \begin{cases} (X[j] - X[i])^2 & \text{if } i \leq j \\ \infty & \text{otherwise} \end{cases}$$

Prove that $M$ satisfies the **Monge property:** $M[i, j] + M[i', j'] \leq M[i, j'] + M[i', j]$ for all indices $i < i'$ and $j < j'$.

**Solution:** Fix arbitrary indices $i < i'$ and $j < j'$ there are two cases to consider.
• If \( i' > j \), then \( M[i', j] = \infty \), and therefore \( M[i, j] + M[i', j'] \leq M[i, j'] + M[i', j] = \infty \). (The left side \( M[i, j] + M[i', j'] \) might also be infinite.)

• Suppose \( i' \leq j \). Then \( i < i' \leq j < j' \), so all four entries \( M[i, j], M[i', j'], M[i, j'] \), \( M[i', j] \) are finite. To simplify notation, define four new variables \( a = X[i], b = X[j], x = X[i'], y = X[j'] \). Because the array \( X \) is sorted, we have \( a < x \) and \( b < y \). Straightforward algebra now implies

\[
(M[i, j] + M[i', j']) - (M[i, j'] + M[i', j])
= ((b - a)^2 + (y - x)^2) - ((y - a)^2 + (b - x)^2)
= (b^2 - 2ab + a^2 + y^2 - 2xy + x^2) - (y^2 - 2ay + a^2 + b^2 - 2bx + x^2)
= -2ab - 2xy + 2ay + 2bx
= 2(x - a)(b - y)
< 0.
\]

In both cases, we conclude that \( M[i, j] + M[i', j'] \leq M[i, j'] + M[i', j] \), as required. ■

(c) **[Extra credit]** Describe an algorithm that finds \( k \) intervals with minimum total squared length that cover \( X \) in \( O(nk) \) time. [Hint: Solve part (a) first, then use part (b).]

**Solution:** Consider the two inner loops in the algorithm \texttt{MinCoverRow}:

```
for j ← 1 to n
    MTS\(i, j\) ← \infty
for i ← 1 to j
    MTS\(i, j\) ← \min\{MTS\(i, j\), MTS\((i - 1, j - 1) + (X[j] - X[i])^2\}\}
```

These four lines find the minimum element of every row in an \( n \times n \) array \( Z \) where

\[
Z[i, j] = MTS(i - 1, j - 1) + (X[j] - X[i])^2
\]

for all indices \( i \) and \( j \). We can write \( Z \) as the sum of two arrays \( Z = R + M \), where

\[
R[i, j] = MTS(i - 1, j - 1) \quad \text{and} \quad M[i, j] = (X[j] - X[i])^2
\]

for all \( i \) and \( j \). Every row in \( R \) is constant, so \( R \) is Monge. We proved in part (b) that \( M \) is Monge. Because \( Z \) is the sum of two Monge arrays, \( Z \) is also Monge.

It follows that the SMAWK algorithm can find the minimum element in every row of \( Z \) in \( O(n) \) time. Replacing the two nested for-loops with SMAWK reduces the running time of \texttt{MinCoverRow} to \( O(nk) \), as required. We do not need to explicitly compute the entire array \( Z \); rather, whenever SMAWK needs an entry \( Z[i, j] \), we can compute it on the fly in \( O(1) \) time. ■

**Rubric:** The same argument cannot be used to improve \texttt{MinCoverCol}. For each index \( j \), the inner loops of that algorithm implicitly find the minimum of each row of the array \( Z[1..k, 1..j] \), where \( Z[i, j] = MTS(i - 1, j - 1) + (X[j] - X[i])^2 \). I believe this array is Monge, but the proof is certainly not as simple as the solution above, and substituting SMAWK for the two for-loops would only improve the running time to \( O(n(n + k)) = O(n^2) \).
2. Describe and analyze an algorithm to determine who wins the “Hello, Sweetie!” game, assuming both players play perfectly.

**Solution:** We start by topologically sorting the input dag $G$, because that’s always the first thing one does with a dag. Topological sort labels the vertices with integers from 1 to $V$, so that every edge points from a lower label to a higher label. Because $s$ is the only source, its label is 1, and because $t$ is the only sink, its label is $V$.

We represent the two players with booleans: $\text{TRUE}$ means the Doctor, and $\text{FALSE}$ means River. For any vertices $d$ and $r$ and any boolean $\text{who}$, let $\text{WhoWins}(d, r, \text{who})$ denote the winning player when the Doctor’s token starts on $d$, River’s token starts on $r$, player $\text{who}$ moves first, and both players play perfectly. We need to compute $\text{WhoWins}(s, t, \text{TRUE})$.

If the game is not over, then the Doctor wins moving first if and only if at least one move by the Doctor leads to a position where the Doctor wins moving second, and the Doctor wins moving second if and only if every move by River leads to a position where the Doctor wins moving first.¹ Thus, the function $\text{WhoWins}$ can be computed by the following recursive algorithm:

```plaintext
WhoWins(d, r, who):
if $d = r$
    return $\text{TRUE}$
else if $d = t$ or $r = s$
    return $\text{FALSE}$
else if $\text{who} = \text{TRUE}$
    return $\bigvee_{v \rightarrow d} \text{WhoWins}(v, r, \text{FALSE})$
else if $\text{who} = \text{FALSE}$
    return $\bigwedge_{v \leftarrow r} \text{WhoWins}(d, v, \text{TRUE})$
```

Thanks to our vertex labeling, we can memoize this function into a $V \times V \times 2$ array, indexed by the variables $d$, $r$, and $\text{who}$ in that order. We can fill the array with two nested for loops, decreasing $d$ in one loop and increasing $r$ in the other, considering both players inside the inner loop. The nesting order of the two for-loops doesn’t matter. Explicit pseudocode appears on the next page.

For any node $v$ in $G$, let $\text{in}(v)$ denote the number of edges entering $v$ (the in-degree of $v$), and let and $\text{out}(v)$ denote the number of edges leaving $v$ (the out-degree of $v$). For almost every pair of vertices $d$ and $r$, our algorithm considers all $\text{out}(d)$ possible moves for the Doctor and then all $\text{in}(r)$ possible moves for River. Thus, the total running time of our algorithm is at most

$$
\sum_{d=1}^{V} \sum_{r=1}^{V} O(\text{out}(d) + \text{in}(r)).
$$

Ignoring the big-Oh constant, we can evaluate this sum in two pieces:

$$
\sum_{d=1}^{V} \sum_{r=1}^{V} \text{out}(d) = \sum_{r=1}^{V} \left( \sum_{d=1}^{V} \text{out}(d) \right) = \sum_{r=1}^{V} E = VE
$$

$$
\sum_{d=1}^{V} \sum_{r=1}^{V} \text{in}(r) = \sum_{d=1}^{V} \left( \sum_{r=1}^{V} \text{in}(r) \right) = \sum_{d=1}^{V} E = VE
$$

Less formally, our algorithm considers all $VE$ pairs (Doctor’s position, River’s move) and all $VE$ pairs (Doctor’s move, River’s position), spending $O(1)$ time on each pair. We conclude that our algorithm runs in $O(VE)$ time.

¹This is the recursive definition of “play perfectly”, for any finite two-player game that cannot end in a draw.
**WHOWINS(V,E):**

label vertices of G in topological order

for $d \leftarrow V$ down to 1
  for $r \leftarrow 1$ to $V$
    if $d = r$
      $\text{WhoWins}[d, r, \text{True}] \leftarrow \text{True}$
      $\text{WhoWins}[d, r, \text{False}] \leftarrow \text{True}$
    else if $d = t$ or $r = s$
      $\text{WhoWins}[d, r, \text{True}] \leftarrow \text{False}$
      $\text{WhoWins}[d, r, \text{False}] \leftarrow \text{False}$
    else
      $\text{WhoWins}[d, r, \text{True}] \leftarrow \text{False}$
      for all edges $d \rightarrow v$
        $\text{WhoWins}[d, r, \text{True}] \leftarrow \text{WhoWins}[d, r, \text{True}] \lor \text{WhoWins}[v, r, \text{False}]$
      $\text{WhoWins}[d, r, \text{False}] \leftarrow \text{True}$
      for all edges $v \rightarrow r$
        $\text{WhoWins}[d, r, \text{False}] \leftarrow \text{WhoWins}[d, r, \text{False}] \land \text{WhoWins}[d, v, \text{True}]$

return $\text{WhoWins}[s, t, \text{True}]$

---

**Rubric:** This is not the only correct solution. One can also explicitly construct the recursion dag, with a vertex for every triple $(d, r, \text{who})$ and edges for each legal move, and then evaluate the recurrence by depth-first search. We can also arguably simplify the recurrence logic by asking whether the first player wins and always NADING the results of recursive calls, instead of alternating between ANDs and ORs.

This solution is more detailed than necessary for full credit. In particular, explicit iterative pseudocode is **not** required for full credit; see the instructions for dynamic programming algorithms in HW1.

No penalty for implicitly assuming that the input graph has more than two vertices. (If the graph has only two vertices, then both players win!)