1. Describe and analyze an algorithm to compute a variable-width histogram with minimum cost for a given set of data values.

Solution: We begin by sorting the input array $S[1..n]$. To simplify the algorithm, we add sentinel values $S[0] = 0$ and $S[n+1] = 1$, although we must remember to exclude these sentinels from the heights of any rectangles.

For any indices i and j, let $\boldsymbol{area(i,j)}$ denote the area of a single rectangle defined by breakpoints $S[i]$ and $S[j]$:

$$area(i,j) := \begin{cases} (j-1)S[j] & \text{if } i = 0 \\ (j-i)(S[j]-S[i]) & \text{otherwise} \end{cases}$$

Now let $\boldsymbol{MinCost(i,\ell)}$ denote the minimum cost of a histogram of the points $S[i..n+1]$ with ℓ rectangles, where the first and last breakpoints are $S[i]$ and $S[n+1]$, respectively. This function obeys the following recurrence:

$$MinCost(i,\ell) = \begin{cases} (area(i,n+1))^2 & \text{if } \ell = 1 \\ \min\left\{(area(i,j))^2 + MinCost(j,\ell-1) \;\middle|\; i \le j \le n+1\right\} & \text{otherwise} \end{cases}$$

To solve the original problem, we need to compute $MinCost(0,k)$. Notice that the recurrence allows rectangles of width (and therefore area) zero.

We can memoize this function into a two-dimensional array $MinCost[0..n+1, 1..k]$. Since each entry $MinCost[i,\ell]$ depends only on entries of the form $MinCost[\cdot, \ell-1]$, we can fill this array in column major order, scanning the columns from right to left in the outer loop, and filling each column in arbitrary order in the inner loop. The resulting algorithm runs in $\boldsymbol{O(n^2k)}$ ***time*** and uses $\boldsymbol{O(nk)}$ ***space***.

```
AREA(S, i, j):
  if i = 0
      return (j − 1)S[j]
  else
      return (j − i)(S[j] − S[i])
```

```
MINCOSTHISTOGRAM(S[1..n], k):
  S[0] ← 0
  S[n + 1] ← 1
  for i ← 0 to n + 1
      MinCost[i, 1] ← AREA(S, i, n + 1)^2
  for ℓ ← 2 to n
      for i ← 0 to n + 1
          MinCost[i, ℓ] ← MinCost[i, ℓ − 1]
          for j ← i + 1 to n + 1
              tmp ← AREA(S, i, j)^2 + MinCost[j, ℓ − 1]
              MinCost[i, ℓ] ← min{tmp, MinCost[i, ℓ]}
  return MinCost(0, k)
```

This is more than enough for full credit, but it's not the best we can do. The space bound can be reduced to $O(n)$ by only maintaining two columns of the memoization array. More significantly, we can improve the running time by a factor of n using the SMAWK algorithm, because the array $M[0..n+1, 0..n+1]$ defined by setting

$$M[i,j] = \begin{cases} area(i,j)^2 + MinCost(j,\ell-1) & \text{if } i \le j \\ \infty & \text{otherwise} \end{cases}$$

is totally monotone. Specifically, we can replace the two loops over i and j, which implicitly compute the minimum element in every row of M, with a single call to the SMAWK algorithm. The resulting algorithm runs in $\boldsymbol{O(nk)}$ ***time*** and uses only $\boldsymbol{O(n)}$ ***space***.

But our claim that M is totally monotone requires a proof! It's actually easier to prove that M has a stronger property. A two-dimensional array is ***Monge***[1] if, for any row indices $i < i'$ and any two column indices $j < j'$, we have $M[i,j] + M[i',j'] \le M[i,j'] + M[i',j]$.

Lemma: *Every Monge array is totally monotone.*

Proof: Let M be a two-dimensional array that is *not* totally monotone. Then there must be row indices $i < i'$ and column indices $j < j'$ such that $M[i,j] > M[i,j']$ and $M[i',j] \le M[i',j']$. These two inequalities imply that $M[i,j] + M[i',j'] > M[i,j'] + M[i',j]$. It follows immediately that M is *not* Monge. □

Lemma: *Let R be the $n \times n$ array defined by setting $R[i,j] := area(i,j)^2$ for all $i \le j$ and $R[i,j] := \infty$ for all $i > j$. Then R is a Monge array.*

Proof: Fix arbitrary row indices $i < i'$ and column indices $j < j'$. If $i' > j$, then $R[i',j] = \infty$, which trivially implies $R[i,j] + R[i',j'] \le R[i,j'] + R[i',j]$, so assume that $i' \le j'$. To simplify notation, we introduce the following variables, all of which are non-negative:

$$\begin{array}{lll} a = i' - i + [i = 0] & b = j - i' & c = j' - j \\ A = S[i'] - S[i] & B = S[j] - S[i'] & C = S[j'] - S[j] \end{array}$$

Then straightforward but tedious algebraic manipulation[2] gives us the following.

$$\begin{aligned} R[i,j] + R[i',j'] &= (a+b)^2 (A+B)^2 + (b+c)^2 (B+C)^2 \\ &= a^2 \left(A^2 + 2AB + B^2\right) \\ &\quad + 2ab \left(A^2 + 2AB + B^2\right) \\ &\quad + b^2 \left(A^2 + 2AB + 2B^2 + 2BC + C^2\right) \\ &\quad + 2bc \left(B^2 + 2BC + C^2\right) \\ &\quad + c^2 \left(B^2 + 2BC + C^2\right) \end{aligned}$$

$$\begin{aligned} R[i,j'] + R[i',j] &= (a+b+c)^2 (A+B+C)^2 + b^2 B^2 \\ &= a^2 \left(A^2 + 2AB + B^2 + 2BC + C^2 + 2AC\right) \\ &\quad + 2ab \left(A^2 + 2AB + B^2 + 2BC + C^2 + 2AC\right) \\ &\quad + b^2 \left(A^2 + 2AB + 2B^2 + 2BC + C^2 + 2AC\right) \\ &\quad + 2bc \left(A^2 + 2AB + B^2 + 2BC + C^2 + 2AC\right) \\ &\quad + c^2 \left(A^2 + 2AB + B^2 + 2BC + C^2 + 2AC\right) \\ &\quad + 2ac \left(A^2 + 2AB + B^2 + 2BC + C^2 + 2AC\right) \end{aligned}$$

The terms in red are all non-negative. It follows that $R[i,j] + R[i',j'] \le R[i,j'] + R[i',j]$. □

Lemma: *Fix an arbitrary integer $\ell > 1$. The $n \times n$ array C defined by setting $C[i,j] := MinCost(j, \ell - 1)$ is a Monge array.*

[1] Monge arrays are named after Gaspard Monge, a French mathematician who was one of the first to consider problems related to flows and cuts, in his 1781 *Mémoire sur la Théorie des Déblais et des Remblais*, which (in context) should be translated as "Memoir on the Theory of the Dirt and the Holes".

[2] I'm sure there's a more elegant way to do this, but why bother?

Proof: Fix arbitrary row indices $i < i'$ and column indices $j < j'$. We immediately have $C[i,j] = C[i',j] = \mathit{MinCost}(j, \ell - 1)$ and $C[i',j] = C[i',j'] = \mathit{MinCost}(j', \ell - 1)$, which implies that $C[i,j] + C[i',j'] = C[i,j'] + C[i',j]$. □

Lemma: *The sum of two Monge arrays is a Monge array.*

Proof: Let X and Y be arbitrary Monge arrays, and let $Z = X + Y$. For all row indices $i < i'$ and column indices $j < j'$, we immediately have

$$\begin{aligned} Z[i,j] + Z[i',j'] &= X[i,j] + X[i',j'] + Y[i,j] + Y[i',j'] \\ &\le X[i',j] + X[i,j'] + Y[i',j] + Y[i,j'] = Z[i',j] + Z[i,j']. \end{aligned}$$ □

We conclude that the array $M = C + R$ is Monge and therefore totally monotone, as claimed. Whew! ■

2. Describe and analyze an algorithm to find the length of the longest palindrome path in a directed acyclic graph with labeled vertices.

Solution: Let $G = (V, E)$ be the input dag, and let $label(v)$ denote the label of any node v.

For any two vertices u and v, let $LPPL(u, v)$ denote the length of the longest palindrome that is a label of a path from u to v in G. (LPPL is a mnemonic for "Longest Palindrome Path Length".) This function obeys the following recurrence:

$$LPPL(u,v) = \begin{cases} 1 & \text{if } u = v \\ -\infty & \text{if } label(u) \neq label(v) \\ \max\left\{ \max_{u \to u'} \max_{v' \to v} \left\{ 2 + LPPL(u', v') \right\},\ 2 \right\} & \text{if } label(u) = label(v) \text{ and } u \to v \in E \\ \max_{u \to u'} \max_{v' \to v} \left\{ 2 + LPPL(u', v') \right\} & \text{otherwise} \end{cases}$$

The first two base cases are straightforward. The third case includes a base case for the single edge $u \to v$, which is necessary for even-length palindromes, but also considers other possible paths from u to v.

To argue that the recurrence is correct, we must show that if there are *no* paths from u to v, then $LPPL(u, v) = -\infty$. If u is a sink, then the outer max is over an empty set of edges, so $LPPL(u, v) = \max \varnothing = -\infty$. Similarly, if v is a source, then $LPPL(u, v) = \max_{u \to u'} \max \varnothing = -\infty$. Otherwise, for every pair of edges $u \to u'$ and $v' \to v$, there is no path from u' to v', so the inductive hypothesis implies that $LPPL(u', v') = -\infty$; thus, $LPPL(u, v) = 2 - \infty = -\infty$ as required.

We can memoize this function into a two-dimensional array $LPPL[1..V, 1..V]$, indexed by the vertices ***in topological order***. Each subproblem $LPPL[i, j]$ depends only on other subproblems $LPPL[i', j']$ with $i' > i$ and $j' < j$, so we can fill the array row-by-row from bottom up (decreasing i), filling each row from left to right (increasing j). The following code implicitly uses the additional base case $LPPL(u, v) = -\infty$ if u follows v in topological order.

```
LongestPalindromePath(V, E):
  index the vertices v_1, v_2, ..., v_|V| in topological order
  for i ← |V| down to 1
      LPPL[i, i] ← 1
      for j ← i + 1 to |V|
          LPPL[i, j] ← −∞
          if label(v_i) = label(v_j)
              for all edges v_i → v_k
                  for all edges v_l → v_j
                      LPPL[i, j] ← max{LPPL[i, j], 2 + LPPL[k, l]}
              if v_i → v_j ∈ E
                  LPPL[i, j] ← max{LPPL[i, j], 2}
  maxLPPL ← 1
  for i ← 1 to |V|
      for j ← 1 to |V|
          maxLPPL ← max{maxLPPL, LPPL[i, j]}
  return maxLPPL
```

Alternatively, instead of topological sort, we could use two nested depth-first searches. The outer loop is just a depth-first search of G; at each vertex of G, we perform an independent depth-first search of the reversed graph $rev(G)$.

```
LongestPalindromePath(G):
  for each node u in depth-first order
      for each node v in reverse depth-first order
          ⋱
```

Translating the for-loops into sums give us the following bound on the time spent in the main loop, ignoring constant factors:

$$\begin{aligned}
\sum_u \sum_v \left(1 + \sum_{u \to u'} \sum_{v' \to v} 1\right) &= \sum_u \sum_v 1 + \sum_u \sum_v \sum_{u \to u'} \sum_{v' \to v} 1 \\
&= V^2 + \sum_u \sum_{u \to u'} \left(\sum_v \sum_{v' \to v} 1\right) \\
&= V^2 + \sum_{u \to u'} \left(\sum_{v' \to v} 1\right) \\
&= V^2 + E^2
\end{aligned}$$

We conclude that the entire algorithm runs in $\boldsymbol{O(V^2 + E^2)}$ ***time***. For the nested depth-first search version of the algorithm, the analysis is similar: For each of the $O(V + E)$ steps of the outer depth-first search, we spend $O(V + E)$ time performing an inner depth-first search, so the overall algorithm runs in $O((V + E)^2) = O(V^2 + E^2)$ time. ■