Robert Owens (rao7utn)
Drainage (Basic HW: Greedy & Dynamic Programming)

This problem requires that one find the longest possible drainage path for a given area where the elevation of the land is modeled by a matrix where the value is the elevation of the land at a given x-y coordinate. It is important to note that for this modeling that water can only drain downhill and that it can only travel North, South, East and West exactly.

My approach is solving this problem is as follows:
First, read input to construct a 2-d array to store the table elevation data provided as input.
Second, call “computeLongestRun” which simply looks for the max length drainage path starting from every cell in the 2-d by calling “computeRun” on each cell. Additionally, in this step, an additional 2-d with the same dimensions in instantiated to memoise/cache longest drainage paths from other possible starting points. Note: that lengths of drainage paths are not actually calculated here, it is just kicking off the recurve method “computeRun” from every possible starting point and returning the max drainage path found after the fact.
Third, “computeRun” which has arguments of the 2-d array, table, representing the land elevation, the cache 2-d array, and i and j integers which represent which elevation cell of table that is in question. If the longest run at that point has already been cached then simply return that value, otherwise consider every direction (N, S, E & W) if it is within the table bounds to access an adjacent cell and that elevation is lower than the current table[i, j] elevation then store a possible path moving in that direction as 1 more than the recursive call to “computeRun” on the adjacent cell. Once this process is finished for every possible direction a best possible path for elevation at table[i, j] is the max of the return recursive calls for traveling in each direction; cache this value by storing it in the cache table then return it.
Fourth, At this point the longest possible drainage path has been found by “computeLongestRun” simply output this value in the appropriate format.

This algorithm runs in $\theta(r \cdot c)$ where r is the number of rows and c is the number of columns in input. Reading input is achieved in $\theta(r \cdot c)$ to read each table entry and create the relevant 2-d array entry. Then computing the longest run is $\theta(r \cdot c)$. The first component of this is calculating the longest path from every table entry which takes $\theta(r \cdot c)$ work to call “computeRun” at each table entry. Then the remaining work done by “computeRun” must be upper bounded by $\theta(r \cdot c)$. Consider that the first call to “computeRun” kicks off a recursive stack such that every cell of the table can be visited to compute a possible run of r*c. This would guarantee that every other “computeRun” will occur in $\theta(1)$ time since the solution to every cell is cached. Now consider that even if the first “computeRun” call does not visit every other cell in the matrix some number of cells m will be computed and cached for lookup in constant time and the remaining r*c-m cell will still need to call a recursive stack to discover their longest possible run. Note that all given cell in the matrix could only need $\theta(r \cdot c)$ recursive call stack space and time to “computeRun” and the actual runtime approaches $\theta(1)$ as the algorithm continues since it comes more and more likely that every “computeRun” call will hit a cached value as the answer to more and more cells is cached. The dominate term is $\theta(r \cdot c)$ which is the runtime for this algorithm.

The grader may look at the files of code uploaded with this submission and thus no pseudocode is required here.