Answer 1:
(a) We can solve the problem by the method of guess and verify.

GUESS: \( T(m, n) = \mathcal{O}(m \log n) \)

VERIFY: \( \mathcal{O}(m \log n) \leq c \cdot m \log n \)

\[
\Rightarrow c \cdot k \log \frac{n}{2} + c \cdot (m - k) \log \frac{n}{2} + a \cdot m \leq c \cdot m \log n
\]

\[
\Rightarrow c \cdot \log \frac{n}{2} (k + m - k) + a \cdot m \leq c \cdot m \log n
\]

\[
\Rightarrow c \cdot m (\log n - \log 2) + a \cdot m \leq c \cdot m \log n
\]

\[
\Rightarrow c \cdot m (\log n - 1) + a \cdot m \leq c \cdot m \log n
\]

\[
\Rightarrow c \cdot m \log n - c \cdot m + a \cdot m \leq c \cdot m \log n
\]

\[
\Rightarrow -c \cdot m + a \cdot m \leq 0
\]

\[
\Rightarrow a \leq c
\]

Since \( c \) exists and does not approach zero, the guess is correct.

(b) Let \( G(x) = \sum_{n=0}^{\infty} a_n x^n \)

We are given the equation: \( a_n = 6a_{n-2} - a_{n-1} \)

\[
\Rightarrow a_n x^n = 6a_{n-2} x^n - a_{n-1} x^n
\]

\[
\Rightarrow \sum_{n=2}^{\infty} a_n x^n = \sum_{n=2}^{\infty} 6a_{n-2} x^n - \sum_{n=2}^{\infty} a_{n-1} x^n
\]

\[
\Rightarrow G(x) - a_1 x - a_0 = 6x^2 \cdot G(x) - x(G(x) - a_0)
\]

\[
\Rightarrow G(x) - 2x - 1 = 6x^2 \cdot G(x) - x \cdot G(x) + x
\]

\[
\Rightarrow (1 + x - 6x^2) \cdot G(x) = 1 + 3x
\]

\[
\Rightarrow G(x) = \frac{1 + 3x}{(1 + x - 6x^2)}
\]

\[
\Rightarrow G(x) = \frac{1}{(1 - 2x)} = \sum_{n=0}^{\infty} 2^n x^n
\]

\[
\Rightarrow a_n = 2^n
\]
Answer 2:
The recurrence relation for the algorithm Alg is the following.
\[ T(n) = 3 \times T\left(\frac{n}{2}\right) + a \times (\log n)^3 \quad \text{if } n > 2 \]
\[ T(n) = b \quad \text{if } n \leq 2 \]
We can solve it by Master’s method.
Let us define a recurrence \( T_1(n) \), such that:
\[ T_1(n) = 3 \times T_1\left(\frac{n}{2}\right) + 1 \quad \text{if } n > 2 \]
\[ T_1(n) = b \quad \text{if } n \leq 2 \]
For this recurrence, we can easily apply the first rule of the masters theorem, when \( f(n) \) is polynomially smaller than \( n^{\log_b a} \)
This gives us,
\[ T_1(n) = \mathcal{O}(n) \quad (1) \]
Next, let us define a recurrence \( T_2(n) \), such that:
\[ T_2(n) = 3 \times T_2\left(\frac{n}{2}\right) + a \times n^{\frac{1}{2}} \quad \text{if } n > 2 \]
\[ T_2(n) = b \quad \text{if } n \leq 2 \]
For this recurrence, we can easily apply the first rule of the masters theorem, when \( f(n) \) is polynomially smaller than \( n^{\log_b a} \)
This gives us,
\[ T_2(n) = \mathcal{O}(n) \quad (2) \]
Since, we can see that
\[ T_1(n) \leq T(n) \leq T_2(n) \]
From equation (1) and (2)
\[ T(n) = \mathcal{O}(n) \]

Answer 3:
Approach: The problem can be solved by applying the concept of divide and conquer. The idea is to first find the middle row \( j \) of the matrix. Then, linearly scan row \( j \) to find its leftmost minimum element (say \( x \)) and the corresponding index (\( k \)). Because of the constraints on the matrix \( M \), the leftmost minimum element for all the rows having index less than \( j \) can only be present at positions \( \leq k \), and all the leftmost minimum element for all the rows having index more than \( j \) can only be present at positions \( \geq k \). Hence, we can recursively call the top left quadrant \( (M[1 \cdot (j-1), 1 \cdot k]) \) and bottom right quadrant \( (M[(j+1) \cdot n, k \cdot m]) \) to recursively solve the problem.
Pseudocode:

Let $M$ be the matrix of size $n \times m$

\[
\text{findLeftmostMinVal}(M): \\
\quad \text{if } n == 1 \text{ then} \\
\quad \quad \text{Find the leftmost minimum element } x \text{ through linear search;} \\
\quad \quad \text{return } x; \\
\quad \text{else} \\
\quad \quad j = \left\lfloor \frac{n}{2} \right\rfloor + 1; \\
\quad \quad \text{Find the leftmost minimum element } x \text{ of row } j \text{ in } M, \text{ and its index } k \text{ using} \\
\quad \quad \text{the linear search;} \\
\quad \quad M_1 = M[1 \cdots (j - 1), 1 \cdots k]; \\
\quad \quad M_2 = M[(j + 1) \cdot n, k \cdots m]; \\
\quad \quad I_1 = \text{findLeftmostMinVal}(M_1); \\
\quad \quad I_2 = \text{findLeftmostMinVal}(M_2); \\
\quad \quad \text{return } I_1, x, I_2; \\
\quad \text{end}
\]

**Complexity analysis:** The recurrence relation for this algorithm is same as given in the problem 1 (a).

\[
T(m, n) = T(k, n/2) + T(m - k, n/2) + a \cdot m \quad \text{for } n > 1 \\
T(m, n) = b \cdot m \quad \text{for } n \leq 1
\]

Following the guess and verify solution of the problem 1(a), it can be shown that the time complexity of the algorithm would be $T(m, n) = \mathcal{O}(m \log n)$

**Correctness:** The algorithm needs to return a list of minimum values for each row of the matrix. The base case is correct since for a single row, we just have to do a linear search to get the minimum. For the recursive case, the algorithm first correctly searches the minimum value at index $k$, in the middle row ($j$) of the matrix (This is correct because we are just linearly searching for the element). Then, the algorithm calls the upper left quadrant ($M[1 \cdots (j-1), 1 \cdots k]$) and lower right quadrant ($M[(j + 1) \cdot n, k \cdots m]$) to find the minimum element corresponding to the rows 1 through $j-1$ and $j-1$ through $n$, respectively. It is not required to search other quadrants of the matrix because of the constraint imposed on the given matrix; if the leftmost minimum value in the $i$-th row of $M$ occur at element $k$, then the leftmost value in the $(i+1)$-th row of $M$ would occur at an index $\geq k$. Hence, the algorithm is correct.
Answer 4:

Approach: One of the solutions for this problem is to store, corresponding to every node, the depth of the node in the tree (or distance of the nodes from the root of the tree). Let us denote the value stored for a node \( u \) as \( DEPTH(u) \). Suppose, we want to find the minimum path length between the nodes \( u \) and \( v \). We are also given a function which returns, given two nodes \( u \) and \( v \), the lowest common ancestor (\( w \)) in constant time. We can easily find the length of the unique path between the nodes \( u \) and \( v \), as follows.

\[
\text{Path-length}(u,v) = DEPTH[u] + DEPTH[v] - 2 \times DEPTH[w]
\]

Pseudocode:

Let \( T \) be the tree

\[
\text{buildDataStructure}(T):
\]

\[
\text{DEPTH} = \text{empty list};
\]

\[
\text{traverseNode}(\text{DEPTH}, \text{root}(T), 0);
\]

\[
\text{return DEPTH};
\]

We can traverse the tree using either breadth-first search or the depth-first search. The procedure \( \text{traverseNode} \) (depth-first search) here is defined as follows. Let \( x \) be a node in the tree, \( d \) be the distance between \( \text{root}(T) \) and \( x \), and \( \text{weight}(u,v) \) be the weight of the edge connecting the nodes \( x \) and \( y \).

\[
\text{traverseNode}(\text{DEPTH}, u, d):
\]

\[
\text{DEPTH}[u] = d;
\]

\[
\text{for each child } v \text{ of } u \text{ do}
\]

\[
\text{traverseNode}(\text{DEPTH}, v, d + \text{weight}(u,v));
\]

\[
\text{end}
\]

\[
\text{return } u;
\]

The procedure for getting the path length between \( u \) and \( v \) is given by:

\[
\text{pathLength}(u,v):
\]

\[
w = \text{LCA}(u,v);
\]

\[
\text{pathlength} = \text{DEPTH}[u] + \text{DEPTH}[v] - 2 \times \text{DEPTH}[w];
\]

\[
\text{return pathlength};
\]
**Complexity analysis:** Since the procedure buildDataStructure is a depth-first search, it has the time complexity of $O(n)$. Also, for storing the depth values, corresponding to every node, we can use a hash table. Hence, the space complexity would also be $O(n)$. The procedure pathLength runs in constant time. This is because the runtime of pathLength depends on retrieving values of nodes from DEPTH (constant time) and on the runtime of LCA($u,v$) (given to run in constant time).

**Correctness:** The recursive procedure which computes the length from root($T$) to all nodes starts with the premise that the distance from root($T$) to a given node $u$ is $d$. For any child $v$ of $u$, the distance from root($T$) to $v$ is therefore $d + \text{weight}(u,v)$. The distance from root($T$) to itself is zero, and the recursion terminates on the leaf nodes. Hence, by induction the procedure buildDataStructure, computes the path lengths correctly.