1. \[ M=2 \]
\[
/ \%
/ \%
/ \%
/ \%
\]
KEY: \( m = \text{min value}, \ M = \text{max value} \)
\[
/ \%
/ \%
/ \%
/ \%
\]
\[ m=2 \quad m=-1 \]
\[
/ \%
/ \%
/ \%
/ \%
\]
\[ M=2 \quad M=4 \quad M=-1 \quad M=6 \]
\[
/ \%
/ \%
/ \%
/ \%
\]
\[ -6 \quad 2 \quad 4 \quad -2 \quad -1 \quad -3 \quad 6 \quad 1 \]

2. \[ M>=2 \]
As soon as we see that the third leaf has a value of 4, we know there’s no point looking at more leaves in that subtree, since player 2 will not select a tree with value \( \geq 4 \) when there is one of value 2 available.
\[
/ \%
/ \%
/ \%
/ \%
\]
\[ m<=2 \quad m<=-1 \]
Similarly, as soon as we know that subtree ”-1 -3” has a max value of -1, we know that player 2 will never choose a tree with value larger than -1.
\[
/ \%
/ \%
/ \%
/ \%
\]
\[ M=2 \quad M>=4 \quad M=-1 \]
However, player 1 has already found a subtree with value 2, so there is no need to look any further.
\[
-6 \quad 2 \quad 4 \quad -1 \quad -3 \quad 6 \quad 1 \]

Total nodes visited: 11

3. Iterative deepening search is just repeated depth-first search on larger and larger trees, where each tree has a maximum depth greater than the previous one (trees are constructed from the original search tree by chopping off all subtrees below a certain depth).

4. “Breadth-first” iterative deepening would be no different from regular breadth-first search, since BFS proceeds level by level through the tree. (Actually, if you strictly interpret the term “iterative deepening,” it would require that the entire tree be searched from the root each time the depth is increased, which would be less efficient that ordinary BFS.)

5. The Manhattan distance, if measured in the usual way, would sometimes overestimate the cost. For instance, the last move in the example required shifting the “7” off to the right. But its Manhattan distance from the correct position is two. (A modified Manhattan distance could be constructed that would take the wraparound into account.)

6. start One possible ordering is shown.
7. Changing the value "h = 4" to something very large, e.g., "h = 10," will prevent $A^*$ from ever expanding this node, even though the shortest path to a goal node requires going through this node. Thus, overestimating distance to the goal could prevent $A^*$ from achieving optimality.

8. (a) Exhaustive search

This is clearly the worst option. There are $(n^2)!$ ways to arrange $n^2$ numbers in a square. Of course, some of these are reflections and rotations of one another, but even weeding out these still leaves a huge search space. Very few of the elements in the search space have any hope of possessing the magic square property.

(b) $A^*$ search

$A^*$ is guaranteed to find a solution with the fewest moves if we give it an admissible heuristic. But what would an admissible heuristic be? The obvious candidate is to find, for each row and each column, what its sum is and then find out by how much this sum differs from the goal sum of $n(n^2 + 1)/2$. Add all of these up to get the heuristic. Is this admissible? No — a single swap is guaranteed to change the sum of at least two rows or columns, and possibly as many as two rows and two columns, so the heuristic might overestimate distance to the goal. A better heuristic might be to determine the smallest change that is achievable by any swap, but this would be both expensive to calculate (consider all possible swaps) and still inadmissible, since one swap could solve the problem while making large changes in the row and column sums (imagine swapping the 1 and the 9 in the $3 \times 3$ version). In short, if a good heuristic is available, this method might work, but at the moment we have not identified a good heuristic.

(c) Constrained tree search

This appears to be the most promising, since we can see that once we have place, say, the 9 (for the $3 \times 3$ case), we can’t place anything larger than a 5 in any other square in the same row or column as the 9. Each placement of a number rules out many more placements. We might still need to do backtracking, but we will search much less of the tree than exhaustive search.
(d) Genetic algorithm search

Here, the issues are how to perform crossover and mutation. Mutation could be as simple as swapping two adjacent numbers, but crossover is much less clear — given two attempts to form a magic square, how do we combine them? Even supposing that we come up with something, there is no guarantee that this method will find a true magic square — genetic algorithms are a “meta-heuristic” and do not guarantee discovery of an optimal solution.

(e) Hill-climbing

First, in the worst case we might need to examine all possible pairs of the $n^2$ values, so this could be very expensive. Second, we need to ask whether we could get trapped at a local minimum. There’s no obvious answer to this (which is why I would not ask this on an exam!), but I whipped up a Python program to test the method and found that there are configurations that are non-magic but such that swapping any pair of elements will not produce any improvement.

Even so, we might try this method until we get a “near-optimal” square, and then try some sort of random move to perturb the square and then repeat the hill-climbing process. (A random move might be to rotate three randomly chosen elements.) This is no longer pure hill-climbing, but more of a hybrid method.