For algorithm development problems, include all of the steps of the process and not just some of the steps or the final algorithm. This includes both the five broad steps (specifications, avenues of attack, algorithm, termination and correctness, efficiency) and the substeps within each. "Call out ..." means that the content of that step was present in your answer but you should explicitly identify each step. See the posted writeups for examples.
Comments on specific steps —

- Specifications: call out each piece i.e. explicitly identify the task, the input, the output, and what constitutes a legal output.
- Avenues of attack.
  - Known targets should address the brute force algorithm — briefly outline the algorithm (often along the lines of enumerate all the possibilities or do a computation for each input element — this doesn't need to be detailed) and give its running time.
  - Address all of the potentially applicable patterns, even if they seem difficult or clearly not the way to go — this step is about trying out different patterns to see what might fit. For divide-and-conquer, this is both easy split and easy merge.
  - "Framework, not algorithm" means that this step is about translating the pattern for your problem, not about figuring out how to solve the problem. Figuring out how to ensure the right setup for the subproblem (e.g. how to get a black square in each quadrant of the board for tiling) or how to do the recombination of the subproblem solutions is not the task yet — the goal is identifying what needs to be figured out to complete the algorithm.
- Algorithm. Include all the pieces — identify the subproblem (task, input, output), give the base and main cases, and address the top level (initial subproblem, setup, wrapup) even if those parts are often trivial. Also address special cases — a common one that arises is duplicates. This isn't applicable for the tiling problem, but it is possible for the other problems. Do duplicates pose a challenge for the algorithm? Finally, put all of the pieces together into one statement of the algorithm.
- Termination and correctness. Correctness in particular may seem trivial because of course the steps written are correct, but it is still important to have thought through correctness when you wrote the base and main cases down. Also make sure the subproblems cover everything they need to and not more.
- Efficiency. Write and solve the recurrence relation for the running time. (Use the big-Oh for recurrence relations tables to solve.) Can you do better? For full credit, your algorithm should beat the brute force running time identified

and there should be some reflection on whether further improvements are possible/which direction to look in. Do the split and/or merge steps do something that could potentially be sped up?

Comments on specific problems —

- For the watershed problem, water can flow both ways off of a high point — a high point is on the edge of two adjacent watersheds, and this isn't a special case that needs handling or a convention that water always flows left or towards the steeper slope (and what if the adjacent slopes are equally steep?). What does need consideration is a flat surface — adjacent points with the same elevation. What if that's a minimum? What if it isn't?

- Use induction on your base and main cases as written to show correctness. For the tiling problem, arguing correctness of the tiling because $2^n - 1$ is always divisible by 3 is insufficient — three tiles could be in a row and thus can't be covered by an L-shaped tile. (Also that $2^n - 1$ is divisible by 3 needs to be established.)