Recursion

1. If problem is “simple”, solve it directly. (Base case).
2. Reduce problem into one or more subproblems of the same type.
3. Proof is usually by induction.
   (Prove that the base case works, then argue that it works for the subproblems)
4. Time complexity: solve with Master Theorem.

Divide-and-Conquer

1. Find recursive algorithm.
2. Do the recursion, reduce problem size by half.
3. Merge results of subproblems to get the final answer.
4. Time complexity: solve with Master Theorem:
   T(n) = T(subproblems) + T(merge)

Greedy

1. Construct a solution step-by-step by some greedy rule: pick the smallest, greatest, etc.
2. Proof: there are two methods:
   a. Induction: Prove the solution is optimal for any time. Examples: Dijkstra's Algorithm, pizza restaurant problem, etc.
   b. Exchange argument: prove optimal solution is consistent with the greedy rule by contradiction. ie. assume the optimal solution is different and show why it must be the greedy rule instead.

Dynamic Programming

1. Find the recursive formula.
2. Memorize the subproblems using the proper data structure (ex. 2D array).
3. Evaluate the subproblems in a meaningful way (depends on recursion).
4. Time complexity: same as recursion.

Network Flow (mainly for matching problems and scheduling problems)

1. Usually for optimization problems. (we want to minimize cost or maximize benefit)
2. Build a network graph. Find a source node and a sink node. Note that there could be multiple source nodes. Then, assign an edge capacity for each edge. Meet demands if necessary (some flow already happening). Find lower bound (circulation).
3. Formulate the goal as max flow/min cut.
4. Apply Ford-Fulkerson O(edges*flow) or Edmonds-Karp O((edges^2)*log(flow)) algorithm.

Polynomial-Time Reductions (Karp Reductions)

1. Reduce problem A to problem B. (A ≤p B).
2. Given an instance of problem A, construct an instance of B.
3. Prove “yes” in A also implies a “yes” in B & “yes” in B implies “yes” in A.

Complexity Classes:

P: Can solve in polynomial time.

NP: Can verify the answer in polynomial time.

NP-Hard: Hard problems, not necessarily in NP but could be.

NP-Complete: The hardest problems in NP.

Notes: Every NP-Complete problem is NP-hard, but not every NP-Hard problem is NP-Complete.

Theory: if we were able to prove that SAT is in P, then P=NP. If SAT is not in P, then P≠NP. This is because SAT is NP-Complete. The same logic applies to any other known NP-Complete problem.

Practice problem: Given an NxN matrix A, with nonnegative real numbers, does there exist A', a matrix taking the floor/ceiling of each number, that preserves the column/row sums?