**Problem Overview**  
  
The Shortest Common Supersequence (SCS) problem asks for a shortest sequence that has each of given sequences as subsequences. In general, SCS is NP-hard for an arbitrary number of sequences​ (even for a binary alphabet​).  
However, we focus on a special case: each input sequence is obtained from some unknown original sequence by deleting at most symbols. In other words, the sequences are highly related, differing from by at most deletion operations each. This additional structure enables a more efficient deterministic algorithm than the general NP-hard case.  
  
**Key implications of the special structure:**

* Each is a subsequence of . Therefore itself is a common supersequence of all . Moreover, any symbol of that was not deleted by at least one sequence must appear in the SCS (otherwise that sequence would miss a needed symbol). Conversely, any symbol that was deleted from all sequences is not needed in the SCS (no input requires it). Intuitively, the SCS will be “close” to the original , possibly omitting symbols that every dropped, and possibly merging certain identical symbols if no sequence demands them separately.
* Since each differs from by at most deletions, no two sequences can “drift” too far apart in alignment. In fact, one can show that for any alignment of these sequences to , the pointers (positions) in any two sequences never differ by more than at any time. This will heavily constrain the search space for the SCS.

Using these observations, we can develop a dynamic programming algorithm to find the SCS efficiently for fixed . We will first describe the algorithm, then prove its correctness (it indeed finds a common supersequence and it is the shortest), and finally analyse its complexity and optimality.

**Algorithm Description**  
  
**Idea:** We simulate a *multiple-sequence alignment* of all sequences, effectively reconstructing the original sequence (minus universally deleted symbols) in the shortest possible way. We build the supersequence character by character, ensuring at each step that no sequence falls behind by more than unmatched characters. This condition guarantees we never consider alignments that violate the “at most deletions” constraint.

**State definition:** We use a DP (dynamic programming) state defined by a tuple where is how many characters of have been matched (subsequence-wise) so far. When we are in state , it means we have constructed a partial supersequence that each can match up to position Initially, we start at (no characters matched). A final goal state is , meaning has matched all characters of every sequence (so is a common supersequence). We will explore moves from state to state by appending characters to .

**Transitions:** From a given state , consider the set of *next required symbols* for each sequence:

* For each sequence that is not yet fully matched , let  
   be the next character that needs (either 0 or 1, since the sequences are binary).
* Let be the set of distinct characters needed next by the sequences. Since the alphabet is binary, is a subset of (in some cases will be either or , and at most it can be ).

For each choice of output character , the algorithm can *append* to the growing supersequence and advance all sequences that needed :

* For each such that (sequence ’s next required symbol is ), we increment (meaning now matches this new character).
* For sequences that did not need (either or is already fully matched), remains the same – those sequences effectively skip this new character (this character was removed from them).

This transition produces a new state . We only allow the transition if it respects the deletion-limit constraint for each sequence. Specifically, when we append , any sequence that does *not* match will treat it as a deleted symbol from . We must ensure no sequence accumulates more than such skipped symbols. Formally, in any state we require for each that the number of characters added so far that did not match is . If appending would cause some sequence to exceed skipped symbols, that transition is forbidden.

Equivalently, let be the length of the partial supersequence constructed at a given state. For each sequence , characters are matched, so characters have been added that skipped. The condition is for all . We maintain this as an invariant in our DP: we never enter a state violating this. This dramatically prunes the search space.

**DP recurrence:** We perform a breadth-first search over these states (since each transition adds one character, BFS naturally finds the shortest path to the goal). The recurrence can be formulated as:

Here is the current supersequence length (which would be value of current state), and ranges over possible next letters as described. We initialize and seek . Rather than explicitly storing values, we can perform on-the-fly BFS: push the start state into a queue with distance 0, then explore all valid transitions, marking visited states with their distance (supersequence length) until we reach the goal.

**Output construction:** By storing back-pointers or reconstructing from the DP table, we can retrieve the actual supersequence achieving the minimum length. (In BFS, we can store the preceding state and character for each newly discovered state.) The resulting sequence is the SCS.

**Example:** Suppose and , with . These come from some by at most one deletion each. Our algorithm will explore from :

* State : needs 0, needs 1 (). We have two choices to try: append 0 or append 1. Both are within deletion limits (no skips yet).  
   - If we append 0: new state (since matches it, ; skips it, still  
   ). Now , and has skipped 1 char (still okay, ).  
   - If we append 1: new state (vice versa). Symmetrically valid.
* From state : now ’s next need is 1 (its second char), still needs 1. So . Append 1 → state (both match 1). Deletion counts ok (skipped none, skipped one).
* Continue in this fashion... eventually the BFS finds a goal state. In this example, one shortest path yields supersequence 0101 of length 4. (Indeed SCS length is 4.)

Throughout the search, any path that would require a sequence to skip more than 1 character is pruned. This significantly cuts down the possibilities vs. arbitrary interleavings. The final answer 0101 is found and is guaranteed shortest.

**Summary of the algorithm:**

* Set up initial state in a queue.
* Perform BFS over state space. For each state, determine the set of next possible output bits from the sequences’ current positions. For each , compute the next state. If the deletion-limit invariant holds, enqueue the new state if not seen before.
* Stop when the state is reached; reconstruct the path (the supersequence).
* Output the supersequence .

This algorithm is deterministic (no randomness involved) and systematically explores all feasible supersequences in increasing order of length, so the first time we reach the goal, we have a shortest solution.

**Proof of Correctness**