This document is to provide how you would answer some questions from lecture if they were homework or exam questions instead. Notes in red would not appear in the homework submission, but are for commentary. The text below, in black, would be sufficient for full credit if the Edit Distance problem, presented in lecture, were instead a homework question.

Give a clear English description of what you are trying to evaluate; do not refer to the internal state ("the current indices") or *how* the algorithm makes this computation (that will be described later). In general, I would prefer that you give the function a meaningful name in the context of the problem rather than naming it something like "OPT."

Define Edit(i, j) to be the minimum cost to convert $X[1 \ldots i]$ to $Y[1 \ldots j]$. This could alternately be phrased as "…the first i characters of X to the first j characters of Y."

Give a *correct* recursive algorithm. This can either be explicit, such as via a recursive algorithm (shown here) or with iterative pseudo-code. If you provide the latter, you do not need to give the recursive solution separately, and the memoization method and evaluation order should be clear from your pseudo-code.

```
Edit(i, j)
  if i == 0 then
    return j
  else if j == 0 then
    return i
  ins = 1+ Edit(i, j − 1)
  del = 1+ Edit(i − 1, j)
  if X[i] == Y[j] then
    sub = Edit(i − 1, j − 1)
  else
    sub = 1+ Edit(i − 1, j − 1)
  return min(ins, del, sub)
```

Unless stated otherwise, when we ask you for a dynamic programming algorithm in this class, it is sufficient to compute the value or cost of an optimal solution. For example, in Edit Distance, you do not need to provide the sequence of inserts, deletes, and substitutions, but you do need to find the minimum cost of such.

Lastly, give the details of the iterative algorithm: how do you memoize, in which order do you fill in the vector, and how long does your iterative algorithm take? The first two might be implicit if you write your solution iteratively, although in that case, you must explicitly point out the recursive computation.

This can be stored in a two-dimensional vector, Edit$[0 \ldots n, 0 \ldots m]$, where n and m are the lengths of the two input strings X and Y, respectively. This can then be filled in first by the base cases, and then the general cases can be filled in by increasing value of the first parameter, then increasing value of the second. Each recursive case takes $\mathcal{O}(1)$ to fill in, and there are $\mathcal{O}(nm)$ cases, for a total of $\mathcal{O}(mn)$ time.

In a traditional complexity theory / algorithms course, you would prove here that Independent Set is in $\mathcal{NP}$. You do not need to do that in CompSci 260P.

3-SAT(n variables, k clauses)
  for each clause $A \vee B \vee C$ **do**
    Create 3 vertices // one each A, B, C
    Add edges $(A, B), (B, C), (A, C)$
  for each variable x_i **do**
    Connect any "x_i" node to all $\overline{x_i}$ nodes
  return INDEPENDENT SET(G, k)

Explanation: The idea is that each clause forms a clique, from which at most one vertex can be chosen by the Independent Set solver. Because there are k clauses and k cliques, at most one in each will be chosen. The chosen vertex represents a literal to set to be true or false to satisfy the clause. We add edges between literals and their negations so prohibit them from both being chosen.

No false positives: Here, we show that if Independent Set returns true at the end, then this is the correct answer to the 3-Sat instance we are given. a return value of true indicates that there are k vertices that are chosen. Because at most one can be chosen from each clique, and there are k cliques, there must have been exactly one chosen in each. By construction, no literal and its negation was chosen. This means we can set the chosen vertices' associated variables to be the truth value for that literal, satisfying each clause.

No false negatives: Here, we show that if the 3-Sat instance has a satisfying assignment, the graph we create will cause Independent Set to return true; thus, if there is a return value of "false," we know it's right. Suppose the 3-Sat instance has a satisfying assignment. Then each clause is satisfied in the 3-Sat instance, given that truth value assignment. We can choose arbitrarily one of the set-to-true literals in each clause and select its vertex in the corresponding clause. No such chosen vertices have an edge to a chosen vertex in any other clique (because the truth value assignment would not set a variable to both true and false). Thus, the graph has a valid independent set, and it is of size k because one was chosen in each of the k independent cliques. Therefore, Independent Set will return true for that graph with the parameter k.