alignment of $x[1..i], y[1..j], z[1..k]$. Instead of explicitly considering the last "operation" done in the alignment, we simply consider the last column of the alignment itself. The positions of the gaps in this column indicate which operations were done (insert, delete, substitute). We then get a recursive algorithm which solves for the optimal alignment by trying all 7 possible last columns (as listed in the hint), for each we compute its sum-of-pairs cost, and then add that to the optimal alignment of the appropriate prefixes. The recursive algorithm can be turned into a dynamic programming algorithm in the usual fashion, resulting in a 3D table of size $n_x \times n_y \times n_z$, so the overall running time and space usage is $O(n_x n_y n_z)$.

In order to build the alignment at the end, we find it useful to keep track of the length of the optimal alignment. This is simply one more variable to keep track of in the dynamic programming—one which increments by 1 each recursive step (since each recursive step is figuring out what the last column of the alignment should be).

```
def cost(c1,c2):
  cost_ins = 1
  cost_del = 1
  cost_sub = 1
  if c1 == c2:
    return 0
  if c1 == BLANK and c2 != BLANK:
    return cost_ins
  if c1 != BLANK and c2 == BLANK:
    return cost_del
  return cost_sub // if get here, both are non-blank and they aren't equal

def sum_of_pairs(xi,yi,zi):
  return cost(xi,yi) + cost(xi,zi) + cost(yi,zi)

// Takes in a 3-bit number between 0 and 7, and spits out a subset of {0,1,2}
// saying which positions are 1 in the binary representation of the number
def positions_of_ones(n):
  ones = []
  for (i = 0; i < 3 and n != 0; i++)
    if n \% 2 == 1:
      ones.append(i)
    n -= (n \% 2)
```

```
    n = n / 2
  return ones

// First fill in dynamic programming table
// table is a nx x ny x nz array of ints
// aux is a nx x ny x nz array of triples of ints (pointers to elsewhere in the
table[0][0][0] = 0
aux[0][0][0] = null
len[0][0][0] = 0 // length of the optimal alignment
for i = 1 to nx-1
  for j = 1 to ny-1
    for k = 1 to nz-1
      min = infinity    // dummy value
      min_poss = -1 // dummy value
      min_ptr = null // dummy value
      for poss = 1 to 7
        // Try putting non-blanks where the 1s are in the
        // binary representation of poss
        ones = positions_of_ones(poss)
        if (0 in ones and i==0) or (1 in ones and j==0)
          or (2 in ones and k==0):
          continue
        xi = (0 in ones) ? x[i-1] : BLANK
        yi = (1 in ones) ? y[j-1] : BLANK
        zi = (2 in ones) ? z[k-1] : BLANK
        prev_i = i - (0 in ones ? 1 : 0)
        prev_j = j - (1 in ones ? 1 : 0)
        prev_k = k - (2 in ones ? 1 : 0)
        cost = sum_of_pairs(xi,yi,zi) + table[prev_i][prev_j][prev_k]
        if cost < min:
          min = cost
          min_poss = poss
          min_ptr = (prev_i, prev_j, prev_k)
      table[i][j][k] = min
      aux[i][j][k] = min_ptr
      (prev_i, prev_j, prev_k) = min_ptr
      len[i][j][k] = len[prev_i][prev_j][prev_k] + 1
```

```
// Read off optimal alignment from aux
x_idx = nx // 1-based index of current character of x, starts at end
y_idx = ny // ditto y
z_idx = nz // ditto z

alignment = new char[3][len[nx][ny][nz]]
for i = len[nx][ny][nz]-1 down to 0:
  (next_x_idx,next_y_idx,next_z_idx) = aux[x_idx][y_idx][z_idx]
  if x_idx == next_x_idx:
    alignment[0][i] = BLANK
  else:
    alignment[0][i] = x[x_idx - 1] // Remember x is 0-indexed

  if y_idx == next_y_idx:
    alignment[1][i] = BLANK
  else:
    alignment[1][i] = y[y_idx - 1]

  if z_idx == next_z_idx:
    alignment[2][i] = BLANK
  else:
    alignment[2][i] = z[z_idx - 1]
```

CSCI 3104 Problem Set 7 Solutions

ID:

Profs. Grochow & Layer
Spring 2019, CU-Boulder

1. *Multiple string alignment.* As in class, we consider the operations of substitution (including the zero-cost substitute-a-letter-for-itself "no-op"), insertion, and deletion. An alignment of three strings x, y, z (of lengths n_x, n_y, n_z, respectively) is an array of the form:

$$\begin{array}{cccccccccc} x_1 & x_2 & - & x_3 & - & - & x_4 & \dots & x_{n_x} & - \\ - & y_1 & y_2 & - & - & y_3 & y_4 & \dots & - & y_{n_y} \\ z_1 & - & - & z_2 & z_3 & - & - & \dots & z_{n_z} & - \end{array}$$

That is, in the first row x appears in order, possibly with some gaps, in the second row y does, and in the third row z does. We define the cost of such an alignment as follows. The cost of a column

$$\begin{array}{c} x_2 \\ y_1 \\ - \end{array}$$

is the sum-of-pairs cost, e.g., in the preceding example we look at the cost of aligning x_2 with y_1 (a substitution, costing either 0 or 1 depending on whether $x_2 = y_1$ or not), the cost of aligning x_2 with $-$ (a deletion, costing 1), and the cost of aligning y_1 with $-$ (another deletion), for a total cost of $c_{subs} + 2c_{del} = 3$ for this one column of the alignment. The total cost of the alignment is the sum of the costs of its columns. Given three strings x, y, z, the goal of Multiple String Alignment is to find a minimum-cost alignment.

(a) *(15 pts) Give an efficient (polynomial-time) algorithm for finding optimal alignments of three strings. Describe your algorithm in pseudocode and English, and prove an upper bound on its running time; you do not need to prove correctness (but you will only receive full credit if your algorithm is correct!). Hint 1: start with a recursive algorithm, where the recursion has 7 cases depending on what the last column of the alignment looks like:*

$$\begin{pmatrix} x_{n_x} \\ y_{n_y} \\ z_{n_z} \end{pmatrix}, \begin{pmatrix} - \\ y_{n_y} \\ z_{n_z} \end{pmatrix}, \begin{pmatrix} x_{n_x} \\ - \\ z_{n_z} \end{pmatrix}, \begin{pmatrix} x_{n_x} \\ y_{n_y} \\ - \end{pmatrix}, \begin{pmatrix} - \\ - \\ z_{n_z} \end{pmatrix}, \begin{pmatrix} - \\ y_{n_y} \\ - \end{pmatrix}, \begin{pmatrix} x_{n_x} \\ - \\ - \end{pmatrix}$$

Hint 2: Your recursive algorithm will likely not be very efficient; what technique can you use from class to improve its efficiency?

The idea is similar to that for pairwise alignment of strings. Namely, our subproblems will be indexed by triples (i, j, k), which corresponds to the optimal