Name: Trevor Buck

ID: 109081318

CSCI 3104 **Profs. Grochow & Layer**

Problem Set 7 **Spring 2019, CU-Boulder**

Quick links 1a 1b 2a 2b 3 4a 4b

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:

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

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?

```
int align(int n, int total) {
    if (n = |x|-1) return total   // reached the end of the string
    int cost = 0;
    if((x[n] != null) AND (Y[n] != NULL) AND (Z[n] != NULL))   //case 1
                                                                  all are
                                                                  occupied
        if (x[n] = Y[n] = Z[n])  cost = 0
        else if (x[n] = Y[n] OR x[n] = Z[n] OR Y[n] = Z[n])
            cost = 2
        else cost = 3
    else if (x[n] != null AND Y[n] != NULL)   // case 2 x+y occupied
        if (x[n] == Y[n])  cost = 2
        else cost = 3
    else if (Y[n] != NULL AND Z[n] != NULL)   //case 3 y & z occupied
        if (Y[n] == Z[n]) cost = 2
        else cost = 3
    else if (x[n] != NULL AND z[n] != NULL)   // case 4
                                                 x & z occupied
        if (x[n] == Z[n] cost = 2
        else cost = 3
```

* continued on Next page

```
else if (X[n] != NULL)        // case 5 x occupied
    cost = 2
else if (Y[n] != NULL)        // case 6 y occupied
    cost = 2
else if (Z(n]! = NULL         // case 7 z occupied
    cost = 2
return align(n+1, cost + total)
}
```

Note: case 8 would be where none are occupied, but that means cost = 0

English: starting with the first element in each lst, we check to see how many of them are occupied. Depending on that, we go to the correct case. There are 7 cases in our algorithm where at least one list is occupied. All we do now is determine what the cost of each case would be. This depends solely on whether or not the values of the lists are the same. We end by recursing to the next element in each list and updating our total cost. Once we reach the end of the array, we just return our total cost.

Runtime: $\Theta(n^3)$

Name: Trevor Buck
ID: [redacted]

(b) (7 pts) Consider generalizing your approach to part (a) to k strings instead of 3. How many cases would there be to consider in the recursion? What runtime would the algorithm have? You do not need to give the algorithm, but must argue persuasively that your answers are correct, given your answer to part (a).

Cases: 2^k options because each string is either empty, or contains element (ie. two options) however, we don't need to count the case where all strings are empty $\Rightarrow \boxed{2^k - 1}$

Runtime: Loop through each strng $\Rightarrow \boxed{O(n^k)}$

Name: Trevor Buck
ID: 109081318

CSCI 3104 **Profs. Grochow & Layer**
Problem Set 7 **Spring 2019, CU-Boulder**

2. The most efficient version of the preceding approach we know of for three strings takes an amount of time which becomes impractical as soon as the strings have somewhere between 10^4 and 10^5 characters (which are still actually fairly small sizes if one is considering aligning, e. .g, genomes). Because of this, people seek faster heuristic algorithms, and this is even more true for aligning more than three strings. One natural approach to faster multi-string alignment is to first consider optimal pairwise alignments and somehow use this information to help find the multi-way alignment. Here we show that naive versions of such a strategy are doomed to fail.

   Given an alignment of three strings, we may consider the 2-string alignments it induces: just consider two of the rows of the alignment at a time. If both of those rows have a gap in some column, we treat the pairwise alignment as though that column doesn't exist. For example, the alignment of x and y induced by the figure at the top of Q1 would be

$$\begin{array}{cccccccccc} x_1 & x_2 & - & x_3 & - & x_4 & \cdots & x_{n_x} & - \\ - & y_1 & y_2 & - & y_3 & y_4 & \cdots & - & y_{n_y} \end{array}$$

   Note that the column between x_3 and y_3 got deleted because it consisted of two gaps.

Name: Trevor Buck
ID: 109081318

CSCI 3104 **Profs. Grochow & Layer**
Problem Set 7 **Spring 2019, CU-Boulder**

(a) (5pts) (UPDATED) Give an example of three strings x, y, z such that in any optimal alignment of x, y, z, at least one of the pairs (x, y) or (y, z) or (x, z) is *not* optimally aligned. Prove your example has this property. (An example where some optimal alignment of x, y, z does not contain an optimal alignment of x, y—but may contain optimal alignments of x, z and/or y, z—will earn you partial credit.)

X = (A – – D – –)

Y = (– B – – E –)

Z = (– – C – – F)

---

optimal (XYZ) = (ABCDEF) : cost = 12

optimal (XY) = (ABDE) : cost = 8

optimal (XZ) = (ACDF) : cost = 8

optimal (YZ) = (BCEF) : cost = 8

(b) (7 pts) Prove that the gap between the cost of the pairwise optimal alignments and the pairwise alignments induced by an optimal 3-way alignment can be arbitrarily large. More symbolically, let d_{xy} denote the cost of an optimal alignment of x and y, and for an alignment α of x, y, z, let $d_{xy}(\alpha)$ denote the cost of the pairwise alignment of x and y which is induced by α. Your goal here is: for all $c > 0$, construct three strings x, y, z such that $\max\{d_{xy}(\alpha) - d_{xy}, d_{xz}(\alpha) - d_{xz}, d_{yz}(\alpha) - d_{yz}\} > c$ for all three-way alignments α. (Suppose all three strings have length n; how big can you make the gap c as a function of n, asymptotically?)

As pointed out in problem 1, the upper bound limit on '$d_{AB}(\alpha)$' where A and B are one your strings, is $O(n^k)$ which, in this case, $(k=3)$ is $O(n^3)$. Using our pairwise optimal alignments the ~~worst~~ worst case scenario for runtime is $O(kn^2). \approx O(n^2)$. Which means that the gap is $\boxed{O(n^3-n^2)}$

If we take into account that $n > 100{,}000$, or so, we see that this becomes a huge difference

Name: Trevor Buck
ID: 109081318

CSCI 3104
Problem Set 7

Profs. Grochow & Layer
Spring 2019, CU-Boulder

3. (14 pts) Give an efficient algorithm to compute the *number* of optimal solutions to the Knapsack problem. Recall the Knapsack problem has as its input a list $L = [(v_1, w_1), \ldots, (v_n, w_n)]$ and a threshold weight W, and the goal is to select a subset S of L maximizing $\sum_{i \in S} v_i$, subject to the constraint that $\sum_{i \in S} w_i \leq W$. For this problem, you will count the number of such optimal solutions. Your algorithm should run in $O(nW)$ time. If you only give an inefficient recursive algorithm, you can still receive up to 6 pts. (If you are having trouble solving this problem, you may want to start by developing an inefficient recursive algorithm and then seeing how to improve it using techniques from class.)

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	5	5	5	5	5
2	0	60	65	65	65	65
3	0	60	100	165	165	165
4	0	60	100	165	180	220
5	0	60	100	150	180	220

(ie sample output)

(NOT IMPORTANT) HAHA

// w[x] refers to the global array w that stores all of the weights

```
def reconstruct(i, j)
    if i=0 return 0
    if c[i-1, j] = c[i,j]        // Didn't take item i
        int secondpath = 0
        if c[i-1, j-w[i]] == w-w[i]      // secondpath
            secondpath = reconstruct(i-1, j-w[i])
        return secondpath + reconstruct(i-1, j)
    else return reconstruct(i-1, j-w[i])        // took item i
```

Name: Trevor Buck
ID: 109081318

CSCI 3104 **Profs. Grochow & Layer**
Problem Set 7 **Spring 2019, CU-Boulder**

4. Consider the following variant of string alignment: given two strings x, y, and a positive integer L, find all contiguous substrings of length at least L that are aligned (using no-ops = substituting a letter for itself) in some optimal alignment of x and y. Assume the costs of substitution, insertion, and deletion are given by constants $c_{subs}, c_{ins}, c_{del}$, and that the cost of substituting a letter for itself is zero.

   (a) (2 pts) Show that the output here contains at most $O((n - L)^3)$ substrings.

I did part B first ...
in which I have a 'while loop' and
one 'for loop' that has $(n-L)^2$ operations.
This takes me to $O((n-L)^3)$

(b) (10 pts) Give an algorithm that solves this problem. You may use/modify/refer to the pseudocode for Knapsack from the lecture notes.

```
def algorithm(x, y, L)
  int i=0
  While i < len(x)
      count = 0
      r = ""
      d = i
      if len(r) ≤ L :
          for j in range(len(y))
              if x[i] == y[j]
                  r = r + "" + y[j]
                  cont += 1
                  i += 1
              else:
                  if count > 0
                  i = d
                  r = ""
                  count = 0
          if x > 0
              z.append(r)
              r = ""
              i = d
              count = 0
  i = i+1
  return len(z)
```