Name: Jon Abrahamson
ID: 107084898

CSCI 3104
Problem Set 7

Profs. Grochow & Layer
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:

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

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?

Name: Jon Abrahamson
ID: 107084898

CSCI 3104
Problem Set 7

Profs. Grochow & Layer
Spring 2019, CU-Boulder

Pseudocode

```
int  low=0;                                  //minimum total cost column index
int  i=0;                                    //the first column index
int align(i, low):
  int total_cost=0;                          //total cost for column
  while i<=n:
    if((x_i!=null) & (y_i!=null) & (z_i!=null))        //case 1 value in every column [no (-)]
      if(x_i=y_i=z_i):
        total_cost=0;                        //subcase if all entries have the same value
      if((x_i=y_i!=z_i) || (z_i=y_i!=x_i) || (x_i=z_i!=y_i)):   //subcase if 2 values the same and 1 different
        total_cost=2;
      if(x_i!=y_i!=z_i):                     //subcase for all different value
        total_cost=3;
    if((x_i=null) & (y_i!=null) & (z_i!=null)):        //case 2: x is null, y and z are not null
      if(y_i=z_i): total_cost=2;
      if(y_i!=z_i): total_cost=3;
    if((x_i!=null) & (y_i=null) & (z_i!=null)):        //case 3: y is null, x and z are not null
      if(x_i=z_i): total_cost=2;
      if(x_i!=z_i): total_cost=3
    if((x_i!=null) & (y_i!=null) & (z_i=null)):        //case 4:  z is null, x and y are not null
      if(x_i=y_i): total_cost=2
      if(x_i!=y_i): total_cost=3
    if((x_i=null) & (y_i=null) & (z_i!=null)):         //case 5: x and y are null, z is not null
      total_cost=2                           //only one none null value so can't be the same
    if((x_i=null) & (y_i!=null) & (z_i=null)):         //case 6: x and z are null, y is not null
      total_cost=2                           //only one none null value so can't be the same
    if((x_i!=null) & (y_i=null) & (z_i=null)):  //case 7: y and z are null, x is not null
      total_cost=2                           //only one none null value so can't be the same
    if(total_cost<column[i]):
      low=i                                  //update the minimum cost index
      i++
      align(i)                               //recursion call
    else:
      i++                                    //this time there is no update to the minimum total cost
      align(i)                               //recursion call
  return i                                   //return the index of the minimum cost column
```

English Explanation

We begin with initializing variables that for the index of the column, the current index of the column that contains the least cost, and the total cost of the current column to zero. We then begin to calculate the total cost of the current column index. There are seven cases possible and each is described in the comments of the pseudocode above. A few of the cases also have subcases to account for equality between different variables (also described in the pseudocode above). For example, one case is when 2 of the 3 variables are not null or not represented by the (-). However, this still leaves many variants. We must then check to see if these 2 non-null variables are equivalent or not because this would affect the total cost for the column. After the total cost of the column is found, we then compare it to the total cost of the column who currently holds the minimum total cost. If the new total cost is less than the current minimum cost column, then this new column replaces the index of the old minimum cost column. Once every column is checked, the algorithm then returns the index of the column that holds the minimum total cost value. The algorithm is then called recursively to check every column.

Time Complexity

We have a while loop that must iterate over a total of 3 input strings giving it a complexity of $O(n^3)$. The rest of the operations can run in O(1) time. Therefore the time complexity is as follows

$T(n) = O(n^3)+O(1) = O(n^3)$

How Could We Improve The Algorithm?

If we can check each of the cases at once without having to iterate through every single one of them would exponentially decrease the run time. In this algorithm it would decrease the run time from $O(n^3)$ to $O(n^2)$.

Name: Jon Abrahamson
ID: 107084898
Profs. Grochow & Layer
Spring 2019, CU-Boulder

(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 for k strings = $2^k - 1$ cases

IE: for 3 strings, like we solved, there are $2^3 - 1 = 7$ cases
for 4 there are $2^4 - 1 = 15$ cases.

This time a while loop must iterate over k input strings with complexity $O(n^k)$. There are still conditional statements with $O(1)$.

$\therefore T(n) = O(n^k) + O(1) = O(n^k)$

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.

(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.)

I Don't know

(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?)

I don't know

Name: Jon Abrahamson
ID: 107084898

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.)

```
Total_Solutions(N, W){
    B_Max, K_Max, W_Max = 0
    for w=0 to W {B[0,w]=0}
    for k=1 to n {B[k,0]=0}
    for k=1 to n {
        for w=0 to W {
            if w[k] ≤ w {
                take = b[k]+B[k-1,w]
                leave = B[k-1,w]
                B[k,w] = max(take, leave)
                if B[k,w] > B_Max {
                    B_Max = B[k,w]
                    K_Max = k
                    W_Max = w}
            else {
                B[k,w] = B[k-1,w]
                if B[k,w] > B_Max {
                    B_Max = B[k,w]
                    K_Max = k-1
                    W_Max = w}
    if number_sol = 0 {
        overall_Max = B_Max
        number_sol++, del N[k], del W[w]
    else {
        if overall_Max = B_Max {number_sol++, del N[k], del W[w]}
        else {return number_sol}
Total_Solutions(N,W)}}}}
```

As found in class, the time complexity to find the first B[k,w] is $\Theta(nW)$, unfortunately the recursion of my algorithm will increase the run time because the whole solution must ran again until a B_Max less than the first solution is found. So if every solution is the same, thus every solution is optimal, then this algorithm must run at $T(n) = O(nW) \cdot O(nW)$

so $T(n) = O((nW)^2)$

*Refferenced Lecture 6

Name: Jon Abrahamson
ID: 107084898
Profs. Grochow & Layer
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.

If we refer to the algorithm on 4b. we see that it consists of a nested for loop in a while loop. Normally this would result in a time complexity of $O(n^2)$ but we are iterating through 2 input strings making $T(n) = O(n^3)$.

However, because the input is not n, but restricted to $n-L$ steps, then $(n-L)$ would be the steps of the time complexity

$\therefore T(n) = O[(n-L)^3]$

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

```
Contiguous_Substr (string x, string y, int L):
    Output = []
    X = X.noPunctuation.lowercase
    Y = Y.noPunctuation.lowercase
    arrx = x.split()
    arry = y.split()
    i = 0
    while i<len(arrx):
        x = 0
        str = ""
        index = i
        if len(str) <= L
            for j in range len(arry):
                if arrx[i] == arry[j]:
                    str = str + ' ' + arry[j]
                    x++
                    i++
                if x > 0:
                    output.aprend(str)
                    i = index
                    str = ""
                    x = 0
            if x>0:
                output.aprend(str)
                str = ""
                i = index
                x = 0
            i++
    return output
```