Name: Alex Book
ID: 108073300
Collaboration: Jacob Reed, Jamie Foster, Varun Narayanswamy

CSCI 3104, Algorithms **Profs. Hoenigman & Agrawal**
Problem Set 10b (34 points) **Fall 2019, CU-Boulder**

Advice 1: For every problem in this class, you must justify your answer: show how you arrived at it and why it is correct. If there are assumptions you need to make along the way, state those clearly.

Advice 2: Verbal reasoning is typically insufficient for full credit. Instead, write a logical argument, in the style of a mathematical proof.

Instructions for submitting your solution:

- The solutions **should be typed** and we cannot accept hand-written solutions. Here's a short intro to Latex.
- You should submit your work through **Gradescope** only.
- If you don't have an account on it, sign up for one using your CU email. You should have gotten an email to sign up. If your name based CU email doesn't work, try the identikey@colorado.edu version.
- Gradescope will only accept **.pdf** files (except for code files that should be submitted separately on Gradescope if a problem set has them) and **try to fit your work in the box provided**.
- You cannot submit a pdf which has less pages than what we provided you as Gradescope won't allow it.

Name: Alex Book
ID: 108073300
Collaboration: Jacob Reed, Jamie Foster, Varun Narayanswamy

CSCI 3104, Algorithms **Profs. Hoenigman & Agrawal**
Problem Set 10b (34 points) **Fall 2019, CU-Boulder**

Important: This assignment has 1 (Q1) coding question.

- You need to submit 1 python file.
- The .py file should run for you to get points and name the file as following -
  If Q1 asks for a python code, please submit it with the following naming convention - `Lastname-Firstname-PS10b-Q1.py`.
- You need to submit the code via Canvas but the table/plot/result should be on the main .pdf.

Name: Alex Book
ID: 108073300
Collaboration: Jacob Reed, Jamie Foster, Varun Narayanswamy

CSCI 3104, Algorithms **Profs. Hoenigman & Agrawal**
Problem Set 10b (34 points) **Fall 2019, CU-Boulder**

1. (34 pts total) Recall that the *string alignment problem* takes as input two strings x and y, composed of symbols $x_i, y_j \in \Sigma$, for a fixed symbol set Σ, and returns a minimal-cost set of *edit* operations for transforming the string x into string y.

   Let x contain n_x symbols, let y contain n_y symbols, and let the set of edit operations be those defined in the lecture notes (substitution, insertion, and deletion).

   Let the cost of *insert* be c_{insert} and *delete* be c_{delete}, and the cost of *sub* be c_{sub}, except when $x_i = y_j$, which is a "no-op" and has cost 0.

   In this problem, you will implement and apply three functions.

   (i) `alignStrings(x,y,` c_{insert}`,` c_{delete}`,` c_{sub}`)` takes as input two ASCII strings x and y, cost of the operations, and runs a dynamic programming algorithm to return the cost matrix S, which contains the optimal costs for all the subproblems for aligning these two strings.

   (ii) `extractAlignment(S,x,y,` c_{insert}`,` c_{delete}`,` c_{sub}`)` takes as input an optimal cost matrix S, strings x, y, cost of the operations, and returns a vector a that represents an optimal sequence of edit operations to convert x into y. This optimal sequence is recovered by finding a path on the implicit DAG of decisions made by `alignStrings` to obtain the value $S[n_x, n_y]$, starting from $S[0, 0]$.

   When storing the sequence of edit operations in a, use a special symbol to denote no-ops.

   (iii) `commonSubstrings(x,L,a)` which takes as input the ASCII string x, an integer $1 \leq L \leq n_x$, and an optimal sequence a of edits to x, which would transform x into y. This function returns each of the substrings of length at least L in x that aligns exactly, via a run of no-ops, to a substring in y.

   (a) (21 pts) From scratch, implement the functions `alignStrings`, `extractAlignment`, and `commonSubstrings`. You may not use any library functions that make their implementation trivial. Within your implementation of `extractAlignment`, ties must be broken uniformly at random.
   If you plan to create a version that saves the parent information in `alignStrings` itself, then you should break the ties randomly in `alignStrings` instead.

   Submit:

   - A brief paragraph for each function that explains how you implemented it (describe how it works and how it uses its data structures).
   - Your code implementation, with code (the code should be submitted on Canvas)

- The cost matrix S that your code produces on the strings x=`EXPONENTIAL` and y=`POLYNOMIAL` with $c_{insert} = 2$, $c_{delete} = 1$, $c_{sub} = 2$

Solution.
Function: alignStrings
Explanation: The function first initializes a cost matrix of all zeroes, from which point it then fills it row by row. In the nested 'for' loops responsible for iterating through the array, there is an 'if' and 'elif' that take care of filling the zero-th row and column with 0...len(x) and 0...len(y), respectively. The second 'elif' statement corresponds to a no-op, when the letter in x and y are the same, in which case the cost is copied from the S[i-1][j-1] cell, as that cell corresponds to a no-op. The 'else' statement handles the case in which a sub, insert, or delete is needed for alignment. The 'min' function is used to make the best possible choice, using the cell corresponding to each operation summed with the cost of that operation. Once the cost matrix has been filled, it is returned.

Function: extractAlignment
Explanation: Firstly, apologies for the mess of 'if' statements in this function. By the time instructors on Piazza had suggested various other solutions I had already gotten all functions cleanly working and didn't quite have the free time to change it, so please bear with me through this explanation. First, an empty array 'operations' is initialized to store the operations that are made to align the given strings. Indexing variables i and j are then initialized in order to start backtracking at the last (bottom right) value in the given cost matrix. The core of this function is held in a 'while' loop that assures that the backtracking process continues until the cost matrix has been necessarily analyzed. Within the 'while' loop, the first 'if' statement checks to see if the letters from x and y that correspond with the current spot in the cost matrix are equal. If so, no operation was required to align them, and "no-op" is appended to 'operations'. The following 'else' statement handles the case in which a sub, insert, or delete was needed for alignment. First, variables are initialized to store the costs of the various operations by summing the corresponding cells and the costs of the correct operations. The following 'if' and two 'elif' statements handle the cases in which one operation is cheaper than the other two, in which cases i and/or j are decremented appropriately and the correct operation is appended to 'operations'. In the case of a tie, one of the tied operations is randomly selected, i and/or j are decremented appropriately, and the correct operations is appended to 'operations'. Once the backtracking process is complete, 'operations' is reversed then returned (because the 'operations'

had the alignment operations stored from the end of the strings to the beginning).

Function: commonSubstrings
Explanation: First, empty arrays 'ret' (to store all common substrings of at least size 'L') and 'current' (to store the current substring) are initialized. Then, index variables x_i is initialized to keep the correct place in the given string 'x' (the reason for this will be given later in this explanation). The following 'for' loop iterates through the given list of operations 'a'. The 'if' statement inside the loop handles the case where a no-op is encountered in the list of operations, in which the correct letter in 'x' is appended to the current substring and x_i is incremented by 1, as the x_i-th letter was already used. The 'elif' statement handles the case where an insert is encountered in the list of operations, in which the common substring must be over (as an operation was needed for alignment) and is therefore added to 'ret' if it is at least length 'L'. Then 'current' is reset to empty, ready to be filled again. In this case, x_i is not incremented because a letter in 'x' was never used (this is where, in class, we would put a blank/underscore in the source string). The 'else' statement handles the case where a delete or sub was used to align the strings, in which the common substring must be over (as an operation was needed for alignment) and is therefore added to 'ret' if it is at least length 'L'. In this case, x_i is incremented because a delete or sub operation 'uses' the x_i-th letter in 'x'. After the for loops exits, an 'if' statement checks if the current substring is at least length 'L' and if so, appends it to 'ret'. This check is needed because if the list of operations ends with one or more no-ops, there will be a leftover, un-handled substring that may need to be added to the list of common substrings.

Cost matrix produced by alignStrings("EXPONENTIAL","POLYNOMIAL",2,1,2):

```
[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11],
 [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12],
 [3, 2, 4, 6, 7, 8, 9, 10, 11, 12, 13],
 [4, 3, 2, 4, 6, 8, 8, 10, 12, 13, 14],
 [5, 4, 3, 4, 6, 6, 8, 10, 12, 14, 15],
 [6, 5, 4, 5, 6, 7, 8, 10, 12, 14, 16],
 [7, 6, 5, 6, 7, 6, 8, 10, 12, 14, 16],
 [8, 7, 6, 7, 8, 7, 8, 10, 12, 14, 16],
 [9, 8, 7, 8, 9, 8, 9, 10, 10, 12, 14],
 [10, 9, 8, 9, 10, 9, 10, 11, 11, 10, 12],
 [11, 10, 9, 8, 10, 10, 11, 12, 12, 11, 10]]
```

Name: Alex Book
ID: 108073300
Collaboration: Jacob Reed, Jamie Foster, Varun Narayanswamy

(b) (7 pts) Using asymptotic analysis, determine the running time of the call
`commonSubstrings(x, L, extractAlignment( alignStrings(x,y,`c_{insert}`,`c_{delete}`,`c_{sub}`), x,y,`c_{insert}`,`c_{delete}`,`c_{sub}` ) )`
Justify your answer.

Solution.
Since all 3 functions are iterative rather than recursive and any calls to other functions happen simply when passing in a variable, running times of the functions are added in order to determine total running time of that call.

Starting with the inner-most function call, alignStrings itself has a running time of $O(n * m)$, where $n = len(x)$ and $m = len(y)$. This is because the nested 'for' loops are of those dimensions, and the operations inside the loops run in constant time.

Next, extractAlignment itself has a running time of $O(n + m)$. This is because the worst possible case is when two strings contain none of the same characters and the cost of a subsitution is more than the sum of the costs of insertion and deletion. In such a case, the alignment would require deletion of every character from the source and insertion of every target character into the the source, totalling $n + m$ iterations of the 'while' loop, which contains only constant time operations.

Finally, commonSubstrings itself has a running time of $O(n + m)$. Since the 'while' loop in extractAlignment runs $n + m$ times in the worst case, the list of operations it produces is of length $n + m$. The length of the list of operations is what determines the runtime of commonSubstrings because its 'for' loop runs as many times as the length of that list. The operations in the loop run in constant time, so the total running time of commonSubstrings must therefore be $O(n+m)$.

For the total running time, sum the running times of all functions in the call.
$O(n * m) + O(n + m) + O(n + m) = O(n * m)$
Total running time of the call $= O(n * m)$

(c) (6 pts) **Plagiarism detector** - String alignment algorithms can be used to detect changes between different versions of the same document (as in version control systems) or to detect verbatim copying between different documents (as in plagiarism detection systems).

The two song lyrics files for PS10b (see class Canvas) contain lyrics of two different songs in text format. Use your functions from (1a) with $c_{insert} = 1$, $c_{delete} = 1$, $c_{sub} = 1$ to align the text of these two documents. Utilize your **commonSubstrings** function for plagiarism detection. Present the results of your analysis, including a reporting of all the substrings in x of length $L = 10$ or more that could have been taken from y in two columns with the first being the length of the substring and the second being the actual common substring obtained via continuous 'no-op' run.

Briefly comment on whether these songs could be reasonably considered original works, under CU's academic integrity policy.

Solution.

Length	Common Substring
18	"I hear the train a"
12	" it's rollin"
28	"round the bend And I ain't "
26	" since I don't know when "
42	" When I was just a baby my mama told me"
34	" When I hear that whistle blowin'"
19	" I rang my head and"
22	" rich folks eatin' in "
36	" fancy dining car They're probably "
44	" if that railroad train was mine I bet I'd "
43	"n a little farther down the line Far from "
62	" to stay And I'd let that lonesome whistle blow my blues away"

The lengths of the source and target strings are 874 and 896, respectively. The sum of the lengths of all common substrings is 386. So, about 44.16 percent of the source string (x) could have been taken from the target string (y). According to OIT's description of TurnItIn standards, 40-60 percent similarity is flagged as "yellow", warranting at the very least further manual review, at the very worst automatic plagiarism qualification. It stands to reason that these songs could not be justifiably considered original works under CU's academic integrity policy (at least before possible further manual review).