CSCI 3104 Spring 2017 **Problem Set 6**

Matthew Niemiec (06/24)
Mehmet Karaoglu (07/30)

---

1. (60 pts) 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, deletion, and transposition).

   Let the cost of *indel* be 1, the cost of *swap* be 10 (plus the cost of the two *sub* ops), and the cost of *sub* be 10, except when $x_i = y_j$, which is a "no-op" and has cost 0.

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

   (i) `alignStrings(x,y)` takes as input two ASCII strings x and y, 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.

```
alignStrings(x,y) :              // x,y are ASCII strings
   S = table of length nx by ny   // for memoizing the subproblem costs
   initialize S                   // fill in the basecases
   for i = 1 to nx
        for j = 1 to ny
             S[i,j] = cost(i,j)   // optimal cost for x[0..i] and y[0..j]
   }}
   return S
```

   (ii) `extractAlignment(S)` takes as input an optimal cost matrix S 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 $S[0, 0]$.

```
extractAlignment(S) :      // S is an optimal cost matrix from alignStrings
   initialize a            // empty vector of edit operations
   [i,j] = [nx,ny]         // initialize the search for a path to S[0,0]
   while i > 0 or j > 0
```

```
        a[i]  = determineOptimalOp(S,i,j) // what was the optimal choice here?
        [i,j] = updateIndices(S,i,j,a)    // move to next position
    }
    return a
```

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) 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.

Submit (i) a paragraph for each function that explains how you implemented it (describe how it works and how it uses its data structures), and (ii) your code implementation, with code comments.

Hint: test your code by reproducing the `APE` / `STEP` and the `EXPONENTIAL` / `POLYNOMIAL` examples in the lecture notes (to do this exactly, you'll need to use unit costs instead of the ones given above).

All code is provided at the end of the document.

(b) Using asymptotic analysis, determine the running time of the call
`commonSubstrings(x, L, extractAlignment( alignStrings(x,y) ) )`
Justify your answer.

Let $len(x) = m$ and $len(y) = n$. The runtime of the call above is $O(nm)$. Notice how that is the equivalent of saying

```
result1 = alignStrings(x, y)
result2 = extractAlignment(result1, x, y)
commonSubstrings(x, L, result2)
```

Which means that the runtime is $O(alignStrings()) + O(extractAlignment()) + O(commonSubstrings()) =$

$O(max(alignStrings(), extractAlignment(), commonSubstrings()))$.
`commonSubstrings()` goes through and looks at every operation in the *edits* list. The worst case length of this list is all insert and delete operations, which would result in $m + n$ edits. Within the *while* loop is $O(1)$, so therefore the runtime of `commonSubstrings()` is $O(m + n)$. In extract alignment we have a lengthy amount of if statements which run in constant time, though lengthy it may be. the while loop runs from the bottom-right hand corner of the matrix to the top-left hand corner of the matrix. For the same reason as in `commonSubstrings()`, the runtime of `extractAlignment()` is $O(n + m)$. The runtime of `alignStrings()` depends on the nested for loops hidden after the nested `cost()` function. First we create an mxn matrix, and then loop through that. That results in mn calls to `cost()`, which runs in constant time, making `alignStrings()` $O(mn)$. Since $mn \geq m + n \ \forall m, n \geq 2$, the runtime of the long nested function call above is $\underline{O(mn)}$.

(c) Describe an algorithm for counting the number of optimal alignments, given an optimal cost matrix S. Prove that your algorithm is correct, and give is asymptotic running time.

Hint: Convert this problem into a form that allows us to apply an algorithm we've already seen.

Thinking of the matrix as a graph, let each number be a node. We also have an array of nodes. Starting at the top-left corner, we add that node to the first spot in the array. Then we look at all the possible numbers which were valid from that first node in the `extractAlignment` algorithm. We take each of these numbers and add them to the array. Note that it is important that we never remove an element from the array, since we want everything to be over-counted. Moving through the array, we then go to the next element added, and add all of that element's possible nodes. We move to the next node, and the next, etc. until we reach the end of our array, which will happen since no nodes will be added because of the top-right hand node. We then take that array and count the number of times the top-right node appears, and that is the number of paths.

This is correct because it is a breath-first search, so it is guaranteed to find all possible paths. Since we aren't keeping track of which nodes we've queued, every path will be counted exactly once for each path that goes through it. As shown in lecture notes, the runtime of BFS is $O(V + E)$, where V is the number of vertices and E is the number of edges. In this case we have mn vertices. Each vertex has at most 4 edges, so there are asymptotically $O(4mn)$ edges. Therefore the runtime is $O(mn + 4mn) = O(5mn) = \underline{O(mn)}$.

(d) 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 `data_string` files for PS6 (see class Moodle) contain actual documents recently released by two independent organizations. Use your functions from (1a) to align the text of these two documents. 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, and briefly comment on whether these documents could be reasonably considered original works, under CU's academic honesty policy.

First we broke up the data file into words, because we don't care if a sequence of characters is similar. For example, "I went to the store" and "Eli went to Cancun" are not worth comparing, though they have a common substring of characters of length 10. Next we went and calculated the cost matrix. Then we ran `extractAlignment()` 20 times. Every time there was 5 occurrences of a common string greater than 10 words long. According to CU's Honor Policy, plagiarism is "failing to use quotation marks when directly quoting from a source."

And indeed we find, printing out the matches found reveals that several belong to the same sequence of strings. Upon investigating we find the string in $data_string_x.txt$: "exxon mobil is strategically investing in new refining and chemical-manufacturing projects in the united states gulf coast region to expand its manufacturing and export capacity. the companys growing the gulf program consists of 11 major chemical, refining, lubricant and liquefied natural gas projects at proposed new and existing facilities along the texas and louisiana coasts. investments began in 2013 and are expected to continue through at least 2022", and from $data_string_y.txt$: "exxonmobil is strategically investing in new refining and chemical-manufacturing projects in the u.s. gulf coast region to expand its manufacturing and export capacity. the companys growing the gulf expansion program, consists of 11 major chemical, refining, lubricant and liquefied natural gas projects at proposed new and existing facilities along the texas and louisiana coasts. investments began in 2013 and are expected to continue through at least 2022."

The similarities don't end there, but it's not as immediately obvious. In fact, if not for those three sentences, our algorithm might not have detected the similarities.

However, there were other matches that would have raised flags for our algorithm to catch. At any rate, it is safe to say that, by CU's Academic Integrity Policy, one of the two sources failed to use quotation marks and directly quoted from the other.

(e) (10 pts extra credit) Find a different document, from the same organization that produced x, which contains substantial copied text from some other document y, produced by a different organization. Present your results. (This is real detective work!)

The first source is a White House Press release found at https://www.whitehouse.gov/the-press-office/2017/03/22/president-trump-approves-wyoming-disaster-declaration and the second is another non-White-House government agency, found at https://www.fema.gov/news-release/2017/03/22/president-trump-approves-major-disaster-declaration-wyoming. They are both talking about how President Trump approved a disaster declaration for the state of Wyoming due to their severe winter storm. Why can't we get some of that weather? Anyways, there were 10 matches of length 10 or more between the two documents, and they were all in the same paragraph. It is unclear who is responsible for the copying and pasting of text, but there is no doubt that they come from the same source. The identical sequences were:

```
['efforts', 'in', 'the', 'areas', 'affected', 'by', 'a',
     'severe', 'winter', 'storm']
['and', 'straight-line', 'winds', 'from', 'february',
    '6', 'to', 'february', '7,', '2017.']
['federal', 'funding', 'is', 'available', 'to', 'state,',
    'tribal,', 'and', 'eligible', 'local']
['governments', 'and', 'certain', 'private', 'nonprofit', 'organizations',
    'on', 'a', 'cost-sharing', 'basis']
['for', 'emergency', 'work', 'and', 'the', 'repair', 'or',
    'replacement', 'of', 'facilities']
['damaged', 'by', 'the', 'severe', 'winter', 'storm', 'and',
    'straight-line', 'winds', 'in']
['teton', 'county.', 'federal', 'funding', 'is', 'also',
    'available', 'on', 'a', 'cost-sharing']
['basis', 'for', 'hazard', 'mitigation', 'measures', 'statewide.',
    'robert', 'j.', 'fenton,', 'acting']
['the', 'affected', 'areas.', 'additional', 'designations',
    'may', 'be', 'made', 'at', 'a']
```

```
['the', 'state', 'and', 'warranted', 'by', 'the', 'results',
    'of', 'further', 'damage']
```

(f) (10 pts extra credit) Infinite Monkey Theorem: *a monkey hitting keys at random on a typewriter keyboard for an infinite amount of time will almost surely type a given text, such as the complete works of William Shakespeare.* Let's find out!

The `data_MuchAdo_txt` file for PS6 (see class Moodle) contains an ASCII version of Shakespeare's play *Much Ado About Nothing*, Act 1 Scene 2, which will serve as an input string y. Write a function that takes as input the `data_MuchAdo_freqs` file for PS6 (see class Moodle), which gives the frequencies of the ASCII characters in the scene, and an integer n, and outputs a file x that contains n characters drawn randomly but with the given frequencies. Then, using your string alignment functions, determine what value of n is required to produce an overlap of 7 characters. Present your results with a brief discussion about what you learned.

As shown in the code posted below, I ran an experiment. I started with n as the length of the poem. When n is too large there's almost no comparisons between the poem and the random string, just a lot of indels. It also ran a lot faster, since larger matrices take a lot longer to go through. We then ran a loop that created a new string of length n. While that was running, we did some other stuff, checked some email, typed this up, got some coffee, and we are currently still waiting for a match of length 7 to be found. Anyways, that finally ran until the test had been run enough times, and the final result was that $n = \underline{253726}$. This is a lot of characters just to find a similar match of length 7. Those monkeys are going to have to type for a very, very long time before they get the complete works of Shakespeare. $(AA^T)^T$

2. (25 pts) *Vankin's Mile* is a solitaire game played by young wizards on an $n \times n$ square grid. The wizard starts by placing a token on any square of the grid. Then on each turn, the wizard moves the token either one square to the right or one square down. The game ends when the wizard moves the token off the edge of the board. Each square of the grid has a numerical value, which could be positive, negative, or zero. The wizard starts with a score of zero; whenever the token lands on a square, the wizard adds its value to his score. The object of the game is to score as many points as possible, without resorting to magic.

   For example, given the grid, the wizard can score $8 - 6 + 7 - 3 + 4 = 10$ points by placing the initial token on the 8 in the second row, and then moving down, down, right, down, down. (This is not the best possible score for these values.)

   (a) Give an algorithm (including pseudocode) to compute the maximum possible score for a game of Vankin's Mile, given the $n \times n$ array of values as input.

   The pseudocode for our Vankin's Mile algorithm is as follows:

```
def VarkinsMile(A):
    max = -inf.
    for i = 0 to len(A) - 1:
        for j = 0 to len(A) - 1:
            if (i != 0) && (j != 0): // element NOT on upper or left edge of array
               A[i,j] = max(A[i,j], A[i-1,j] + A[i,j], A[i,j-1] + A[i,j])
               if (i == len(A) - 1) && (A[i,j] > max):
                    max = A[i,j]
            else:
                if (i == 0) && (j != 0): // element on upper edge, excluding A[0,0]
                   A[i,j] = max(A[i,j], A[i,j-1] + A[i,j] )
                   if (j == len(A) - 1) && (A[i,j] > max):
                       max = A[i,j]
                else if (j == 0) && (i != 0): // on left edge, excluding A[0,0]
                   A[i,j] = max(A[i,j], A[i-1,j] + A[i,j])
                   if (i == len(A) - 1) && (A[i,j] > max):
                       max = A[i,j]
                else:
                   if A[i,j] > max , max = A[i,j] // A[0,0]
    return max
```

We have developed a sort of a dynamic programming approach to solve the Vankin's Mile problem. We use the original $n \times n$ array that is inputted to our function as the table that we store the solutions to our subproblems in (to save space, we do not allocate a new array). We transform the original array according to a method that will be described below, and when our algorithm terminates, it will have found the maximum score that can be obtained. The basic idea of our algorithm is that we traverse the entire 2-D array, and for each element of the array, we replace it with the maximum score that can be obtained by placing the token on that particular element after a series of placing it (or lack thereof) on other elements. This, however, requires the knowledge of the maximum scores that can be obtained by reaching the element one unit above or one unit to the left (since the only legal movement operations of the game are to go directly to the right or directly below). So, solving for the maximum score of a particular element, which itself is a subproblem of the whole problem, requires us to know the values for at most two other subproblems, and those subproblems also consist of at most two subproblems, and so on. For each element in the array, we compare the element's value to the value above it added to the element, and the value to the left added to the element (note that the values above and to the left might have been changed). We choose the option that gives us the maximum value, and update the variable `max` if `A[i,j]` is on the right edge or bottom edge of the array, since the game can only end if you go off the right or bottom edge. We consider the four possible cases: an element that is *not* on the left or upper edge of the array, an element that is on the left edge (because $j - 1 = -1$ in this case), an element that is on the upper edge ($i - 1 = -1$), and the first element in the array, `A[0,0]`. After we have evaluated every single element in the array, our algorithm returns `max`, which holds the maximum score that can be obtained.

(b) Prove that your algorithm is correct.

We will prove our algorithm by showing that it maintains a certain loop invariant throughout its entire running time. The following statements are true before each iteration of our `for` loop:

i. Our array consists of two regions: any element that lies *above* or to the *left* of a particular element contains the maximum value that can be obtained by placing a token on the original value of that element after a series of placing it (or lack thereof) on other elements. Any element that lies *below* or to the *right* of an element has not been evaluated yet and thus contains the original value stored in that element.

ii. The max variable has been updated to hold the maximum score if `A[i,j]` is on the right or bottom edge.

Initiation: The trivial case is when we're about to evaluate the first element of the array. There is no element to the left or above the first element, and every element to the right or below is just the rest of the entire array which is not evaluated, so the first statement of our loop invariant is trivially correct. The maximum score so far is `-inf.` since we havent evaluated any elements yet.

Maintenance: During each execution of our `for` loop, we update the element that we're currently on with the maximum score that we can obtain up until that element. We consider three choices to replace the current value of the element with: we can choose to keep the current value, replace it with the value to the left added to the current value, or replace it with the value to the right added to the current value - we choose whichever option has a greater value. Then, we check if the value we chose is greater than the current maximum score of the entire game board, and if the element is on the right or bottom edge. If both are true, we replace the variable `max` with the current value, if not, we leave `max` as it is. After the loop finishes and moves on to the next element, the element to the left contains the maximum possible value obtained by landing on that element, and `max` contains the overall maximum score. The remaining elements have yet to be evaluated. Thus, we maintain the loop invariant after each iteration of the for loop.

Termination: After we finish evaluating the last element of the array, every single element to the left and above has been evaluated and the score for it has been found, the `max` variable now contains the maximum score of the *entire* game board, and there are no elements remaining to the right or below the element. We have successfully calculated the maximum score obtainable by playing the Vankins Mile game on the $n \times n$ game board.

We have shown that our loop invariant is correct for the trivial case, is maintained after each iteration of the `for` loop, and is maintained when we terminate. Thus, our algorithm is correct for this problem.

(c) Analyze your algorithm's time and space requirements.

Space requirements: Our algorithm doesn't use any auxiliary data structures to compute the solution; it is an *in-place* algorithm. We transform the original inputted array and store in it the solutions to our subproblems, thus saving space. Hence, the space requirement for our algorithm would be $\underline{O(1)}$.

Time requirements: The running time of our algorithm is $\underline{\Theta(n^2)}$. The outer `for` loop, which goes through every *row* of the inputted array, runs exactly n times, where n = number of rows/columns of the array. For each iteration of the outer loop, the inner loop also runs n times (for each column), giving a total running time of $\Theta(n^2)$.

3. (15 pts) A simple graph (V, E) is bipartite if and only if the vertices V can be partitioned into two subsets L and R, such that for every edge $(i, j) \in E$, if $i \in L$ then $j \in R$ or if $i \in R$ then $j \in L$.

   (a) Prove that every tree is a bipartite graph.
   Hint: Try a proof by contradiction, and think about cycles.

   We first give one of the definitions of a bipartite graph: A graph is bipartite *if and only if* it contains no odd cycles.

   We will prove that every tree is a bipartite graph by using proof by contradiction. First, we assume that there exists a tree - which is just a graph with no cycles (acyclic) - that has edges E and vertices V, and that the tree is *not* a bipartite graph. Then, using the definition of a bipartite graph stated above, the tree will need to contain at least one odd cycle. An odd cycle in the context of graph theory is a cycle which has an odd number (n) of vertices. Then, we imply that the graph will contain at least one cycle. However, this presents a contradiction to our original assumption that our graph is a tree - a graph which has at least one cycle *cannot* be a tree, and vice-versa. Therefore, we hold that there cannot exist a tree which isn't bipartite and that **every tree must be a bipartite graph**.

   (b) Adapt an algorithm described in class so that it will determine whether a given undirected graph is bipartite. Give and justify its running time.

   Determining if bipartite: We will modify the Breadth-First search algorithm that was described in in class, and change it so that it will determine whether a given undirected graph is bipartite. A graph is bipartite if its vertices can be partitioned into two parts or sets (hence the name bipartite) so that if one of the vertices of an edge is in one set, the other should be in the other set. We change the original coloring method for Breadth-First search so that instead of coloring unvisited vertices only one color, we will use two colors - a different color for the two different sets. We are going to color unvisited vertices with the opposite color of the current vertex, and after we dequeue the vertex from the queue, we are also going to check every adjacent vertex to see if it has the opposite color as the current vertex. If an adjacent vertex has the same color as the current, then our graph *cannot* be bipartite because we will have two adjacent vertices in the same set, which violates the rules of bipartite graphs. If our algorithm finds two adjacent vertices of the same color, it will return and print "Not bipartite graph." If it doesn't encounter any adjacent vertices of same color, it will return normally.

Running time: The running time of our algorithm will essentially be the same as the running time of the Breadth-First search algorithm because the code that we add to it doesn't asymptotically affect it. We are still visiting each vertex $u \in V$ and checking all of their adjacent vertices v, which is $|E|$ at the most. The only things we really added is deciding if two adjacent vertices are in different sets and we changed how we do the coloring of unvisited vertices. The running time, thus, is $\Theta(V + E)$ assuming we're using an adjacency list representation for our graph.

4. (15 pts) Prof. Dumbledore needs your help to compute the in- and out-degrees of all vertices in a directed multigraph G. However, he is not sure how to represent the graph so that the calculation is most efficient. For each of the three possible representations, express your answers in asymptotic notation (the only notation Dumbledore understands), in terms of V and E, and justify your claim.

   (a) An *edge list* representation. Assume vertices have arbitrary labels.

   Computing the in degrees and out-degrees of the vertices of a graph using an edge list representation will take $\Theta(VE)$ time, since we would need to check each element of the list (each element lists a particular edge of the graph) for both in degrees and out degrees, and we would need to check them for each vertex. $VE + VE = 2VE = \underline{\Theta(VE)}$.

   (b) An *adjacency list* representation. Assume the vector's length is known.

   An adjacency list representation consists of an array, *Adj*, of $|V|$ lists. In adjacency list representations of directed graphs, for each $u \in V$, the list in *Adj*$[u]$ contains all of the vertices v such that an edge originates at u and ends at v. In order to compute the out-degrees of all vertices in the the adjacency list, we need to check each u in *Adj* and count all of the elements in the list *Adj*$[u]$ (find the length of *Adj*$[u]$) because each element of *Adj*$[u]$ represents an edge going out from u. The sum of all of the adjacency lists for directed graphs will be equal to $|E|$, so the **out-degrees** will be computed in $\underline{\Theta(V + E)}$ time.

   To compute the in-degrees, we will need to count the total amount of times a particular v occurs in *each* adjacency list, and we need to do this for *each* possible vertex. This means that we need to check every element in each *Adj*$[u]$ ($|E|$ elements) $|V|$ times. We will find the **in-degrees** in $\underline{\Theta(VE)}$ time.

   Summing the two running times, we will have a total running time of $\underline{\Theta(VE) + \Theta(V + E)}$.

   (c) An *adjacency matrix* representation. Assume the size of the matrix is known.

   Computing the **in-degrees** and **out-degrees** of all vertices using an adjacency matrix representation will both take V^2 time each, and computing both will thus have a total running time of $2V^2$ or $\underline{\Theta(V^2)}$. Computing the in-degrees of one of the vertices requires summing up the values contained in each row (equivalently, going through each column) for every single row, which would mean checking every single element of the array (the array has V^2 elements).

Similarly, computing the out-degrees would require summing up each element in each row for every single column, which is essentially the same thing as checking every single element.

Therefore, the total running time will be $\underline{\Theta(V^2)}$.

```
from random import random, randint
import numpy as np

COST_INDEL = 1
COST_SWAP = 10
COST_SUB = 10


def print_matrix(matrix):
    print(np.matrix(matrix))


# This is going to return the minimal cost of aligning 2 strings
# For parameters, string1 is the string on top and string2 is the string on the
# side. Doesn’t matter for the algorithm, but it is useful for testing purposes
def align_strings(string1, string2):
    # This is a nested function that calculates the cost. It is helpful because
    # it keeps the bulk of the code clean and has access to all of this
    # function’s local variables, seen as global variables in this context
    def cost(i, j):
        costs = []

        # The top left corner of the matrix
        if i == j == 0:
            return 0

        # Costs of indel operations
        if i > 0:
            costs.append(matrix[i-1][j] + COST_INDEL)
        if j > 0:
            costs.append(matrix[i][j-1] + COST_INDEL)

        # Cost of sub
        # Doesn’t take into account the space at the beginning
        if i > 0 and j > 0:
            if string1[j-1] == string2[i-1]:
                costs.append(matrix[i-1][j-1])
            else:
                costs.append(matrix[i-1][j-1] + COST_SUB)

        # Cost of swap
        # Also doesn’t take into account the space at the beginning
        if i > 1 and j > 1:
            swap = (string1[j-1] != string2[i-2] + string1[j-2] != string2[i-1])
            swap *= COST_SUB
            swap += matrix[i-2][j-2] + COST_SWAP
            costs.append(swap)
        # Return the minimum of all the available ops
        return min(costs)

    # Create an mxn matrix based on the lengths of the strings
    matrix = [[0]*(len(string1)+1) for _ in range((len(string2)+1))]

    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            matrix[i][j] = cost(i, j)
    return matrix


# Returns a vector the represents an optimal solution that converts x into y
def extract_alignment(matrix, string1, string2):
    high_bound = max((x for y in matrix for x in y)) + 1
    vector = []
    i = len(string2)
    j = len(string1)
    while i > 0 or j > 0:
        choices = []
        # Potential case of an indel
        if i > 0:
            choices.append(matrix[i-1][j])
        else:
            choices.append(high_bound)
        # Potential case of the other indel
        if j > 0:
            choices.append(matrix[i][j-1])
        else:
```

```
        choices.append(high_bound)
    # Potential case of a sub
    if i > 0 and j > 0:
        choices.append(matrix[i-1][j-1])
    else:
        choices.append(high_bound)
    # Case where there was potentially a swap
    if i > 1 and j > 1:
        diff = matrix[i][j] - matrix[i - 2][j - 2]
        subs = (string1[j-1] != string2[i - 2]) + (string1[j - 2] != string2[i-1])
        subs *= COST_SUB
        subs += COST_SWAP
        # Check if it's possible for a swap to have occurred
        if diff == subs:
            choices.append(matrix[i-2][j-2])
        else:
            choices.append(high_bound)
    else:
        choices.append(high_bound)
    # Find the minimum possible previous number
    small = min(choices)
    # Find the number of times that min occurs
    routes = choices.count(small)
    # Make an int decision, which is a number between 1 and the times min occurred
    decision = int(random()*routes) + 1
    temp = -1
    # Each time we encounter min we decrement decision
    # When decision is 0, we choose that instance of min/path
    while decision > 0:
        temp += 1
        if choices[temp] == small:
            decision -= 1

    # Case where decision is the insert a space
    if temp == 0:
        vector.append('INS')
        i -= 1
    # Case where decision is to delete the letter. Append deletion symbol
```

```
        elif temp == 1:
            vector.append('DEL')
            j -= 1
        # Case of swap. If a match, insert accordingly
        elif temp == 2:
            # Letters align - case of no-op
            if small == matrix[i][j]:
                vector.append('|')
            else:
                vector.append('SUB')
            i -= 1
            j -= 1
        # Case where decision is to swap the two letters
        elif temp == 3:
            vector.append('SWAP')
            vector.append('SWAP')
            j -= 2
            i -= 2
    # Switch the vector order since it was initially backwards
    vector.reverse()
    return vector


# The function that takes in the list of edits and the first string
# and returns the total number of instances where the two strings were the same
def common_substrings(string1, edits, length):
    i = 0
    count = 0
    current_streak = 0
    # Go thought the length of the list of edits
    while i < len(edits):
        if edits[i] == '|':
            current_streak += 1
            # When there are 10 or more noops in a row
            if current_streak == length:
                # Print the specific case where the match occurs with plagiarism
                output = string1[i-length+1:i+1]
                if isinstance(output, list):
                    print(output)
                count += 1
                current_streak = 0
        else:
            current_streak = 0
        i += 1
    return count


# This is code that we use to test our functions above. The results are
# What was given in class
s1 = "polynomial"
s2 = "exponential"
MIN_COMMON_LENGTH = 3
cost_matrix = align_strings(s1, s2)  # In-class exercise. The result should be 3
print_matrix(cost_matrix)
fix = extract_alignment(cost_matrix, s1, s2)
print(fix)
common_strings = common_substrings(s1, fix, 3)
print(common_strings)

# This is for part c. We renamed the files so that they weren't really long to type
reading = open('data_string_x.txt', 'r')
data_x = reading.read().split()
reading.close()
reading = open('data_string_y.txt', 'r')
data_y = reading.read().split()
reading.close()

text_matrix = align_strings(data_x, data_y)
text_fix = extract_alignment(text_matrix, data_x, data_y)
text_lengths = common_substrings(data_x, text_fix, 10)
print("Next:", text_lengths)

# read in the files, make them lowercase, split them up into words, and only take into
# account the first 2000 so that our algorithm doesn't take a really long time to run
reading = open('whitehouse.txt', 'r')
test1 = reading.read().lower().split()[:2000]
```

```
reading.close()
reading = open('fema.txt', 'r')
test2 = reading.read().lower().split()[:2000]
reading.close()

# Run the same tests again
text_matrix = align_strings(test1, test2)
text_fix = extract_alignment(text_matrix, test1, test2)
text_lengths = common_substrings(test1, text_fix, 10)
print("Next:", text_lengths)

# We read in the last files
reading = open('muchAdo_txt.txt', 'r')
ado = reading.read()
reading.close()
reading = open('muchAdo_freqs.txt', 'r')
ado_freq = reading.read()
reading.close()
monkey_string = ""
for line in ado_freq.split('\n'):
    line = line.split()
    if len(line) == 3:
        line[0] = "' '"
    monkey_string += line[0][1]*int(line[-1])
# monkey_string = ''.join(sample(monkey_string, len(monkey_string)))
monkey_length = len(monkey_string)
n = monkey_length
ado_lengths = 0
runs = 0
while not ado_lengths:
    print("Nope")
    test_string = ""
    for _ in range(n):
        test_string += monkey_string[randint(0, monkey_length-1)]
    # print(test_string)
    ado_matrix = align_strings(ado, test_string)
    ado_fix = extract_alignment(ado_matrix, ado, test_string)
    ado_lengths = common_substrings(ado, ado_fix, 7)
    runs += 1
print("Finally! n =", n*runs)
print("Number of matches:", ado_lengths)
```