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).

(b) Using asymptotic analysis, determine the running time of the call

```
commonSubstrings(x, L, extractAlignment( alignStrings(x,y) ) )
```

Justify your answer.

- We define nx as the length of the first string, and ny as the length of the second string.
- alignStrings creates a matrix size $nx \times ny$, and fills this matrix using a nested loop, one running nx times, and the other running ny times. Since we loop through the whole thing regardless, we have $O(nx * ny$ and $\Omega(nx * ny)$, thus the asymptotic runtime is $\Theta(nx * ny)$
- extractAlignment finds an optimal path in the matrix starting at the end, going back to the beginning of the matrix. For each path, we will traverse some fraction of the matrix, but never all of it based on our way of traversal(up, left, diagonal once, or diagonal twice). Therefore our worst case would be inserting or deleting all the letters in one of the strings and vice versa. Therefore we have a runtime of $O(max(nx, ny))$.
- commonSubstrings only loops through the x string and the a list of optimal ops. Since the size of $a \leq len(x)$, it loops at most len(x) times. Therefore the runtime here is $O(nx)$.
- So for the big call commonSubstrings(x, L, extractAlignment(alignStrings(x,y))), the asymptotic runtime is $\Theta(nx*ny)+O(max(nx, ny))+O(nx) = \Theta(nx*ny)$

(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.

- I don't know.

(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.

- After running our algorithm, these are the substrings we came up with:
- " is strategically investing in new refining and chemical-manufacturing projects in the "
- " gulf coast region to expand its manufacturing and export capacity. the companys growing the gulf "
- " consists of 11 major chemical, refining, lubricant and liquefied natural gas projects at proposed new and existing facilities along the texas and louisiana coasts. inves "
- " these jobs will have a multiplier effec"
- " that service these new "
- We found that a good amount of this White House press release was plagiarized.
- It is pretty clear that this would not be considered an original work by CU boulder because of the fact that, after counting the number of similar strings, it turned out to be around 20% plagiarized. Which, by the standards of CU, is considered plagiarism.

(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!)

(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.

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.

   - For Vankin's mile, we start at a position (x,y). From this starting point, we can make a new matrix size $(n - x) \times (n - y)$ instead of $n \times n$.
   - In this new "score" matrix, we will update the values based on the max of the original board's indexes.
   - Each time, we will check the element above and the element to the left, take the max and update the "score" matrix value.
   - Dynamically memoizes each max score at any given index in the matrix.
   - Function takes in the starting point that the wizard has decided.

```
#take in the starting point, Orig board, size of Orig board.
maxVankin(x,y,Orig,n):
   xL = list(n-x)
   xY = list(n-y)


    #2d matrix for scores at each index
    scores = [xL, yL]

    #initialize first index of score board to cost of starting point
    scores[0][0] = Orig[x][y]

    #initialize columns
    for x -> n
     scores[x][0] = scores[x-1][0] + Orig[x][0]

     #initialize rows
    for y -> n
     scores[0][y] = scores[0][y-1] + Orig[0][y]


    #fill in the rest of the board based on (x-1) and (y-1)
    for x -> n
     for y -> n
         scoreAbove = scores[x-1][y] + Orig[x][y]

        scoreLeft = scores[x][y-1] + Orig[x][y]

         theMax = max(scores[x-1][y-1], scoreAbove, scoreLeft)

         scores[x][y] = theMax


    #find the max in the scores array on the last row or last column

    max = 0

    for x -> n
```

```
        if (scores[x][n] > max):
            max = scores[x][n]

    for y -> n
        if (scores[n][y] > max):
            max = scores[n][y]


return max
```

(b) Prove that your algorithm is correct.

- Our problem begins at the starting box, creating a smaller $n \times m$ matrix.
- Each time through the loop, we fill in one box of the scores matrix. This specific square is defined by the max of the boxes before it, up and to the left. This feature gives us the optimal substructure. We compute one box, which later help us define other boxes. itemSince our algorithm greedily chooses the best score for each box, the scores matrix will tell us the score at that corresponding element in the Orig board.
- Lastly, we have a final loop to choose the max score out of the best possible scores at each box.

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

- This algorithm has one nested for loop that runs from $x \to n$ and $y \to n$, which is $O(n^2)$. We can't put a lower bound on it because our starting point can vary, which would vary the number of operations in this loop.
- This function also uses a (n-x) by (n-y) matrix to store the scores of each path. Again we can't put a lower bound on it because x and y vary. Therefore it is $O(n^2)$ space as well.

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.

- Proof: Given a tree T, T is bipartite. Assume, for the sake of contradiction, that T is a tree and it is NOT bipartite ($P \wedge \neg Q$) where P is T is a tree and Q is T is bipartite.
- If a graph is not bipartite it has at least one odd cycle.
- Proof: Given vertex sets V_1 and V_2, if you start walking from V_1, in order to end up in the same place in V_1, you must walk an even number of vertices. Therefore, if you take an odd number of steps you will end up at a vertex in V_2, which contradicts the definition of bipartite.
- Given the fact that if a graph is not bipartite is has at least one odd cycle, it would follow that T has at least one odd cycle because it is not bipartite (due to original contradictory statement). However, this is a contradiction, because, by definition, trees do not have cycles. Because we have arrived at a contradiction, the original statement must be true.

Hint: Try a proof by contradiction, and think about cycles.

(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.

- A good way to find out if a graph is bipartite is to modify Breadth First Search(BFS) to mark vertices in a graph with a certain color. The algorithm will traverse the graph the same way BFS does but it will color each node it visits neighbors the opposite color of the current node it is visiting. The reason for this is because it is impossible to color a bipartite graph where two vertices in the same set are different colors. This is because if a graph is bipartite it will not have any odd cycles. If there is an odd cycle that node will share an edge with another vertex in the same set and they will result in different colors. The algorithm will start at the root and place that node in set A and place its neighbors in set B. If BFS reaches a neighbor node that is the same color as the current node, then the graph is not bipartite. This is because if we find a neighbor node that is the same color as the current node this means that we have two of the same nodes that are different colors in different sets, indicating that there is an odd cycle in the graph. Psuedocode below:

```
isBipartiteBFS(node):
 make queue Q
    mark node visited
    color node RED
```

```
count = 0
 while length(Q) != 0:
     if count % 2 = 0:
         color = BLUE
        else:
         color = RED
     current = head of Q
        mark adjacents of current visited
        if adjacent.color == current.color:
         print "Graph not bipartite"
         break
        add adjacents to Q
        mark adjacents color
```

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.

   - Given an edge list representation, it will take $\Theta(V)$ to compute either in- degree or out- degree for each vertex. An algorithm that does this would simply loop through the edge list, while incrementing an auxiliary array, which has length of the largest numbered vertex. The algorithm would increment the index that corresponds to the vertex number for each of the first numbers in each of the ordered pairs. This calculates out- degree. To calculate in degree, simply increment based on the second number in each of the ordered pairs. Because this loop will always start at the beginning of the edge list and will always terminate at the end, this algorithm is $\Omega(V)$ and $O(V)$, making it $\Theta(V)$ for calculating either in or out degree for each vertex.
   - Note: While this algorithm has a relatively fast runtime it uses a very large amount of space. This is because if a node in the graph has a value of 4,000,000 for example and the rest of the nodes are much smaller (1,2,3,4) it needs an auxiliary array that is 4,000,000 in length. This is a very inefficient use of space.

- Another algorithm that uses less space but has a slower runtime is an algorithm that instead uses a red-black tree to store each vertex. It loops through the edge list and stores each vertex with no duplicates. Then it traverses the tree and increments an auxiliary array based on an array of vertices created by the red-black tree. Lastly, you would loop through the new array and the edge list, incrementing corresponding auxiliary arrays of in and out degrees. This has an asymptotic runtime of $\Theta(E \log V)$, but less space requirement than the first algorithm.88

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

- The runtime of finding in- or out- degrees of vertices will be $\Theta(V + E)$. This algorithm will do basically the same thing as the edge list algorithm. It will loop through the array of vertices and increment an auxiliary array while looping through the linked list of each respective vertex. Because the algorithm has to V total outer loops and E total inner loops the run time is $\Theta(V + E)$ for calculating either in or out degrees of all vertices.

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

- The asymptotic runtime for an algorithm that calculates the in or out degrees of all vertices is $\Theta(V^2)$. This is because the algorithm must loop through the each row of the matrix and increment an auxiliary array based on if there is a one or a zero(or another value if graph is weighted). Because there are V columns and V rows, there are V^2 values to consider. Another way to look at it is there must be an outer for loop that moves the inner loops down one column each time. The inner loop will do V checks and the outer loop will run V times. Regardless of the input both for loops will run V times making it both $\Omega(V^2)$ and $O(V^2)$. Therefore, the runtime is $\Theta(V^2)$.

```
import random

def resolveTiesandMin(l):
    nameList = ["Swap", "Sub", "Indel1", "Indel2"]
    #
    # print l
    ties = []
    minIndex = 0
    for i in range(0, 4): #find min
```

```
        #print "L of ",i," is ", l[i]
        if (l[i] < l[minIndex]):
            minIndex = i
    #print minIndex

    for i in range(0,3): #
        for j in range(i,3):
            #print(i,j)
            if(j!=i and l[i]==l[j] and l[i] <= l[minIndex]):
                if(nameList[i] not in ties):
                    ties.append(nameList[i])
                if (nameList[j] not in ties):
                    ties.append(nameList[j])

    if(ties):
        if "Sub" in ties:
            return "Sub"
        else:
            randomTie = random.choice(ties)
            #print("random ",randomTie)
            return randomTie
    else:
        #print("single min= ",nameList[minIndex])
        return nameList[minIndex]


def commonSubstrings(x, L, a):
    #creates common substrings from x->y
    while(L<1 or L>len(x)):
        newl = int(input("Enter a number between 1 and " +
        str(len(x)) + "\n"))
        L = newl

    subList = list()
    substr = ""
    subchars = list()

    '''print(a)
```

```
print(len(a))
print(len(x))
'''
#for i in range(0,len(x)):

i = 0
while(i < len(x)): #compute cost by looping through s and
appending correct letter to subchars
    if (i >= len(a)):
        i += 1
    elif(a[i] == "NO-OP"):
        subchars.append(x[i])
        i += 1
    elif(a[i] == "Swap"):
        #a.insert(0, " ")
        i += 2
    else:
        subchars.append(" ")
        i += 1



run = False
for char in subchars: #append each subchar to a substring
list that is used for printing the common substrings
    if(char != " "):
        substr = substr + char
        if (char != subchars[-1]):
            run = True
        else:
            run = False
    else:
        if (substr != "" and len(substr) >= L):
            subList.append(substr)
        substr = ""

    if (char == subchars[-1] and run == False):
        if(substr != "" and len(substr) >= L):
```

```
                subList.append(substr)

def extractAlignment(S,x,y):
    #Callculates the shortest and optimal paths.
    nx = len(x)
    ny = len(y)

    i = nx
    j = ny

    #print(len(S[0]))

    a = list()

    #print("i=",i," j=",j)

    while(i>0 or j>0):
        #print("i=",i," j=",j, " S[i][j]=",S[i][j])
        tieList = list()


        swap = 0 #initialize values based on dynamic programming
        algorithm
        sub = 0
        indel1 = 0
        indel2 = 0

        if(i>=2 and j>=2): #assign values to swap, sub, indel1,
        indel2, based on the value of previously determined
        values
            swap = S[i-2][j-2]
        if (i >= 1 and j >= 1):
            sub = S[i-1][j-1]
        if(i>=1 and j >=0):
            indel1 = S[i - 1][j]
        if(j>=1 and i >=0):
            indel2 = S[i][j-1]
```

```
# if((sub == S[i][j])):
#     a.append("NO-OP")
#     print "Q"
#     print i
#     i = i-1
#     j = j-1
# elif(swap == sub and  sub != -1):
#     a.append("Sub")
#     print "z"
#     print i
#     i = i - 1
#     j = j - 1
if(j <= 0):
    #print("special Indel1")
    a.append("Indel1")
    i = i-1
elif(i <= 0):
    #print("special Indel2")
    a.append("Indel2")
    #print a
    j = j-1
else:
    tieList.append(swap)
    tieList.append(sub)
    tieList.append(indel1)
    tieList.append(indel2)
    #print("TIE LIST = ", tieList)
    testString = resolveTiesandMin(tieList)

    if testString is "Sub" and sub is S[i][j]: #loop

 through to create a which will contain the list of edits
                                                #check to
                                                see which
                                                string is
                                                in the
                                                test
```

```
                                                    string and
                                                    append
                                                    that type
                                                    of edit
                a.append("NO-OP")
            else:
                a.append(testString)
            if(testString == "Swap"):
                a.append("Swap")
                i = i-2
                j = j-2
            elif (testString == "Sub"):
                i = i-1
                j = j-1
            elif (testString == "Indel1"):
                i = i-1
            elif (testString == "Indel2"):
                j = j-1



    a.reverse()
    return a

def alignStrings(x,y):
    #fill the matrix of optimal costs.
    nx = len(x)+1
    ny = len(y)+1

    S = []
    for i in range(0, nx): #create matrix and initialize with all
    #zeroes
        new = []
        for j in range (0, ny):
            if(i == 0):
                new.append(j)
            elif(j == 0):
                new.append(i)
```

```
            else:
                new.append(0)
        S.append(new)


    if(x==y):
        return S

    for i in range(1,nx): #actually calculates the dynamic
    programming table by checking the necessary previous values
    such
    as i-2,j-2 and i-1,j-1
        for j in range(1,ny):
            #print("I=",i, "J=", j)
            costofSub = 1
            costofIndel = 1
            costofSwap = 3
            if(x[i-1] == y[j-1]):
               costofSub = 0

            if(i == 1):
                doSub = S[i-1][j-1] + costofSub
                doIndel1 = S[i-1][j] + costofIndel
                doIndel2 = S[i][j-1] + costofIndel
                costweChoose = min(doSub, doIndel1, doIndel2)
                #print("doSub=", doSub, " doIndel1=", doIndel1,
                doIndel2=", doIndel2)
                #print(costweChoose)
            else:
                doSwap = S[i-2][j-2] + costofSwap
                doSub = S[i - 1][j - 1] + costofSub
                doIndel1 = S[i - 1][j] + costofIndel
                doIndel2 = S[i][j - 1] + costofIndel
                costweChoose = min(doSwap, doSub, doIndel1,
                doIndel2)
                #print("doSwap=", doSwap, " doSub=", doSub, "
                doIndel1=", doIndel1, " doIndel2=", doIndel2)
                #print(costweChoose)
```

```
            S[i][j] = costweChoose


    print "[",
    print '-, ',
    for c in y:
        print c,',',
    print"]"

    for i in range(len(S)):
        if(i==0):
            print"-",
        else:
            print x[i-1],
        print S[i]


    return S

def main():

    s1 = "the white house office of the press secretary for
    immediate rel" #entire string was difficult to fit in
    s2 = "exxonmobil plans investments of $20 billion to exp"

    commonSubstrings(s1, 10, extractAlignment((alignStrings(s1,
    s2)), s1, s2))
    print s2
if __name__ == "__main__":
    main()
```