1. (60 points) 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 (substitution, insertion, and deletion).

   Let the cost of *indel* be 1 and the cost of *sub* be 2, 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,x,y)` takes as input an optimal cost matrix S, strings x, y, 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,x,y) :  // S is an optimal cost matrix from alignStrings
   initialize a            // empty list of edit operations
   [i,j] = [nx,ny]         // initialize the search for a path to S[0,0]
   while i > 0 or j > 0
        a.prepend(determineOptimalOp(S,i,j,x,y)) // what was an optimal choice?
        [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), x,y ) )`
Justify your answer.

(c) 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 PS5 (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 10 of the longest substrings in x of length $L = 20$ 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.

(i):

`alignStrings function:`

This function populates the 2D Vector(cost table) with the optimal costs to convert string1 into string2. After the execution of this function optimal costs will be populated. The data structure of 2D array is used to store the 2 dimensional costs of converting string1 to string2. The nested for loop iterate through the 2D array and calls cost function for each cell. This function populates the cell with the optimal cost calculated through dynamic programming technique. For every index i, j of cost table the cell cost table[i][j] contains the optimal value to convert string1[:i] to string2[:j].

`extractAlignment function:`

This function back tracks the optimal path from cost table and store it in the result list. After the execution of this function the sequence of edit operations will be populated in the result list. The data structure used is list, which stores strings on each index. Each string is a name of operation i.e Update, Insert, Delete No-op etc. The while loop iterate through the last index of 2D array of cost table and calls determine optimal operation function. This function calculates the optimal back path from the current index and stores it in the result list. Once determine optimal operations updates the result list, update indices is called to update the indices backward.

`commonSubstrings function:`

This function returns all the substrings of string1 which aligns with substrings of string 2, exactly via sequence of no-ops. Input L is taken from user and passed as a paramter in the function in a variable "length". The function contains nested for loops. Outer for loop is to make sure that all the substrings of length less than L are stored in the list result. Inner loop iterates through all the edit sequence operations to find the substrings. The data structure used to store the substrings is list. Each index of list contains a substring of string1 less than length 'L', which aligns with string 2 with operations of no-ops.

(ii):

```
import numpy as np


def cost(cost_table, string_x, string_y, i, j):
    if i is 0:
        return j   # if length of string 1 is zero, cost will be length of string 2
    elif j is 0:
        return i   # if length of string 2 is zero, cost will be length of string 1
    elif string_x[j - 1] == string_y[i - 1]:    # if both characters of string 1 and 2 are equal
        return cost_table[i - 1][j - 1]      # then cost will be equal to the cost of upper diagonal cell
    else:
        return min(cost_table[i][j - 1], cost_table[i - 1][j], cost_table[i - 1][j - 1]+1) + 1
        # if the characters of both strings are unequal, cost will be equal to the minimum of previous step plus cost of update, insert, delete operation



def align_strings(string_x, string_y):
    nx = len(string_x)   # length of string1
    ny = len(string_y)   # length of string2
    rows = ny+1
    cols = nx+1
    cost_table = np.zeros([rows, cols])  # table initialization from zero

    for i in range(0, rows):
        for j in range(0, cols):
            cost_table[i][j] = cost(cost_table, string_x, string_y, i, j)  # populate table from cost function of edit operations
    return cost_table


def common_substrings(string_x, length, operations):
    result = list()    # list of substrings less than length L
    for i in range(1, length):
        for j in range(0, len(operations)):
            if "No op!" in operations[j]:
                k = j
                count = 0
                res_string = ""
                while count < i and k < len(operations):
                    if "No op" in operations[k]:
                        tokenize = operations[k].split(" ")
                        position = tokenize[len(tokenize)-1]
                        res_string = res_string+string_x[int(position)-1]
                        count = count + 1
                        k = k + 1
                    else:
                        break
                if count == i:
                    result.append(res_string)

    return result


def determine_optimal_operation(cost_table, operations, string_x, string_y, xj, yi):  # back track to find the optimal decision from current cell of cost table
    if xj != 0 and yi != 0:
        if string_x[xj-1] == string_y[yi-1]:  # if elements of both strings are equal than the operation will be no op
            operations.append("No op! For "+string_x[xj-1]+" at position "+str(xj))
            return 1
        else:
            if cost_table[yi][xj] == cost_table[yi - 1][xj - 1]+2: # if cost of current cell is 2 units more than the upper left diagonal cell than operation wil
                operations.append("Update operation! For "+string_x[xj-1]+" to "+string_y[yi-1])
                return 2
            elif cost_table[yi][xj] == cost_table[yi][xj-1]+1: # if cost of current cell is 1 units more than the left cell than operation will be delete
                operations.append("Delete operation! Delete "+string_x[xj-1]+" from string 1 at position "+str(xj))
                return 3
            else:   # if cost of current cell is 1 units more than the upper cell than operation will be delete
                operations.append("Insert operation! Insert "+string_y[yi-1]+" in string 1 at position "+str(xj))
                return 4
    else:
        if xj == 0: # if cost of current cell is 1 units more than the upper cell than operation will be delete
            operations.append("Insert operation! Insert " + string_y[yi - 1] + " in string 1 at position "+str(xj))
            return 5
        elif yi == 0:  # if cost of current cell is 1 units more than the upper cell than operation will be delete
            operations.append("Delete operation! Delete " + string_x[xj - 1] + " from string 1 at position "+str(xj))
```

```
            return 6
        else:
            return -1  # return -1 if both indices are reached to zero


def update_indices(check, yi, xj):
    if check == 1 or check == 2:   # if No-op or Update operation has been performed, then update the indices to the upper left diagonal position
        return [yi-1, xj-1]
    elif check == 3:     # if delete operation has been performed, then update the indices to the left cell position
        return [yi, xj-1]
    elif check == 4:    # if Insert operation has been performed, then update the indices to the upper cell position
        return [yi-1, xj]
    elif check == 5:    # if delete operation has been performed, then update the indices to the left cell position
        return [yi-1,xj]
    elif check == 6:      # if Insert operation has been performed, then update the indices to the upper cell position
        return [yi, xj-1]


def extract_alignments(cost_table, string_x, string_y):
    rows = len(string_y)  # length of string 2
    cols = len(string_x)  # length of string 1
    operations = list()   # vector in which the sequence of edit operations will be added in reverse order of strings
    [i, j] = [rows, cols]

    while i > 0 or j > 0:
        check = determine_optimal_operation(cost_table, operations, string_x, string_y, j, i)  # function which determines the optimal operation at a specific in
        if check == -1:
            break
        [i, j] = update_indices(check, i, j) # update the strings indices depending on the execution of determine_optimal_operation function, variable check is u

    return operations  # returns optimal sequence of edit operations


def print_operations(operations):
    for element in operations:
        print(element)


print("Press 1 to input strings. ")
print("Press 2 to read strings from file.")
choice = input("Enter choice: ")   # Takes input from user on console to decide whether the strings will be from file or input from the user
if choice == str(1):            # if user inputs 1 then
    x = input("Enter first string : ")  # input string 1 from console
    y = input("Enter second string: ")  # input string 2 from console
else:   # if user decide to consider string1 and string2 from input test files
    f = open("csci3104_S2018_PS5_data_string_x.txt", "r",encoding="utf8")   # open file as string 1
    if f.mode == "r":
        x = f.read()
    f = open("csci3104_S2018_PS5_data_string_y.txt", "r",encoding="utf8")   # open file as string 2
    if f.mode == "r":
        y = f.read()


table = align_strings(x, y)   # calls align_strings function
print("AlignStrings Function")
print("Table S : ")
print(table)  # print table S
print("")
print("ExtractAlignment Function")
operations = extract_alignments(table, x, y)  # calls function extract alignments
print_operations(operations)
print("")
operations.reverse()    # reverse the sequence of edit operations to feed in commonSubstring function

print("CommonSubstring Function")
l = input("Input L for common substring function : ")   # input L to find the No-Op substrings
substrings = common_substrings(x, int(l), operations)
print("Sub strings that aligns with a run of no-ops are: ")
print(substrings)
```

```
Alexiss-MacBook-Pro:New alexgarcia$ python test.py
Press 1 to input strings.
Press 2 to read strings from file.
Enter choice: 1
Enter first string : STEP
Enter second string: APE
AlignStrings Function
Table S :
[[ 0.  1.  2.  3.  4.]
 [ 1.  2.  3.  4.  5.]
 [ 2.  3.  4.  5.  4.]
 [ 3.  4.  5.  4.  5.]]

ExtractAlignment Function
Delete operation! Delete P from string 1 at position 4
No op! For E at position 3
Update operation! For T to P
Update operation! For S to A

CommonSubstring Function
Input L for common substring function : 2
Sub strings that aligns with a run of no-ops are:
['E']
Alexiss-MacBook-Pro:New alexgarcia$
```

(b):

Asymptotic Analysis:

AlignString: The time complexity of alignStrings is O(nx * ny), because of nested for loop through each index of cost table. Where nx is length of string1 and ny is the length of string2.

ExtractAlignment: The time complexity of extractAlignment is O(ny+nx), because the maximum no of operations will be deletion of all characters of string where nx is length of string1 and ny is the length of string2.

CommonSubstring: The time complexity of commonSubstring is O(N*len(A)*N) where A is the vector of sequence of edit operations.

Considering L $\leq$ nx
Time complexity = O (nx*nx*len(A)* (nx+ny) * nx * ny)
$= O(nx^3 * ny * (nx * ny) * L)$

L = length of edit operation sequence

(c):

Using the data files provided on Moodle. Ee can go through our Python code and see an analysis of our 10 longest sub strings in x of length 20 by running our files with the functions above. Below you can see an image of the output of analysis.

Strings:

> 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.
> investments began in 2013 and are expected to continue through at least 2022.
>
> these jobs will have a multiplier effect, creating many more jobs in the
>
> that service these new

Based on CU's academic policy, This shows excessive plagiarism. It cant be considered original work.

, 'cal, refining, lubr', 'al, refining, lubri', 'l, refining, lubric', ', refining, lubrica', ' refining, lubrican', 'refining,
lubricant', 'efining, lubricant ', 'fining, lubricant a', 'ining, lubricant an', 'ning, lubricant and', 'ing, lubricant and ', '
ng, lubricant and l', 'g, lubricant and li', ', lubricant and liq', ' lubricant and liqu', 'lubricant and lique', 'ubricant and
liquef', 'bricant and liquefi', 'ricant and liquefie', 'icant and liquefied', 'cant and liquefied ', 'ant and liquefied n', 'nt
and liquefied na', 't and liquefied nat', ' and liquefied natu', 'and liquefied natur', 'nd liquefied natura', 'd liquefied natu
ral', ' liquefied natural ', 'liquefied natural g', 'iquefied natural ga', 'quefied natural gas', 'uefied natural gas ', 'efied
natural gas p', 'fied natural gas pr', 'ied natural gas pro', 'ed natural gas proj', 'd natural gas proje', ' natural gas projec
', 'natural gas project', 'atural gas projects', 'tural gas projects ', 'ural gas projects a', 'ral gas projects at', 'al gas pr
ojects at ', 'l gas projects at p', ' gas projects at pr', 'gas projects at pro', 'as projects at prop', 's projects at propo',
' projects at propos', 'projects at propose', 'rojects at proposed', 'ojects at proposed ', 'jects at proposed n', 'ects at prop
osed ne', 'cts at proposed new', 'ts at proposed new ', 's at proposed new a', ' at proposed new an', 'at proposed new and', 't
proposed new and ', ' proposed new and e', 'proposed new and ex', 'roposed new and exi', 'oposed new and exis', 'posed new and e
xist', 'osed new and existi', 'sed new and existin', 'ed new and existing', 'd new and existing ', ' new and existing f', 'new a
nd existing fa', 'ew and existing fac', 'w and existing faci', ' and existing facil', 'and existing facili', 'nd existing facili
t', 'd existing faciliti', ' existing facilitie', 'existing facilities', 'xisting facilities ', 'isting facilities a', 'sting fa
cilities al', 'ting facilities alo', 'ing facilities alon', 'ng facilities along', 'g facilities along ', ' facilities along t',
 'facilities along th', 'acilities along the', 'cilities along the ', 'ilities along the t', 'lities along the te', 'ities along
 the tex', 'ties along the texa', 'ies along the texas', 'es along the texas ', 's along the texas a', ' along the texas an', 'a
< ng the texas and', 'long the texas and ', 'ong the texas and l', 'ng the texas and lo', 'g the texas and lou', ' the texas and
 oui', 'the texas and louis', 'he texas and louisi', 'e texas and louisia', ' texas and louisian', 'texas and louisiana', 'exas
 and louisiana ', 'xas and louisiana c', 'as and louisiana co', 's and louisiana coa', ' and louisiana coas', 'and louisiana coa
st', 'nd louisiana coasts', 'd louisiana coasts.', ' louisiana coasts. ', 'louisiana coasts. i', 'ouisiana coasts. in', 'uisiana
 coasts. inv', 'isiana coasts. inve', 'siana coasts. inves', 'iana coasts. invest', 'ana coasts. investm', 'na coasts. investme'
, 'a coasts. investmen', ' coasts. investment', 'coasts. investments', 'oasts. investments ', 'asts. investments b', 'sts. inves
tments be', 'ts. investments beg', 's. investments bega', '. investments began', ' investments began ', 'investments began i', '
nvestments began in', 'vestments began in ', 'estments began in 2', 'stments began in 20', 'tments began in 201', 'ments began i
n 2013', 'ents began in 2013 ', 'nts began in 2013 a', 'ts began in 2013 an', 's began in 2013 and', ' began in 2013 and ', 'beg
an in 2013 and a', 'egan in 2013 and ar', 'gan in 2013 and are', 'an in 2013 and are ', 'n in 2013 and are e', ' in 2013 and are
 ex', 'in 2013 and are exp', 'n 2013 and are expe', ' 2013 and are expec', '2013 and are expect', '013 and are expecte', '13 and
 are expected', '3 and are expected ', ' and are expected t', 'and are expected to', 'nd are expected to ', 'd are expected to c
', ' are expected to co', 'are expected to con', 're expected to cont', 'e expected to conti', ' expected to contin', 'expected
to continu', 'xpected to continue', 'pected to continue ', 'ected to continue t', 'cted to continue th', 'ted to continue thr',
'ed to continue thro', 'd to continue throu', ' to continue throug', 'to continue through', 'o continue through ', ' continue th
rough a', 'continue through at', 'ontinue through at ', 'ntinue through at l', 'tinue through at le', 'inue through at lea', 'nu
e through at leas', 'ue through at least', 'e through at least ', ' through at least 2', 'through at least 20', 'hrough at least
 202', 'rough at least 2022', 'ough at least 2022.']

2. (10 points) Shadow is trying to figure out all the potential paths from the bank he is trying to rob to the party's hideout, given by the network below. Help him calculate the number of paths from node 1 to node 14.

   Hint: assume a "path" must have at least one edge in it to be well defined, and use dynamic programming to fill in a table that counts number of paths from each node j to 14, starting from 14 down to 1.

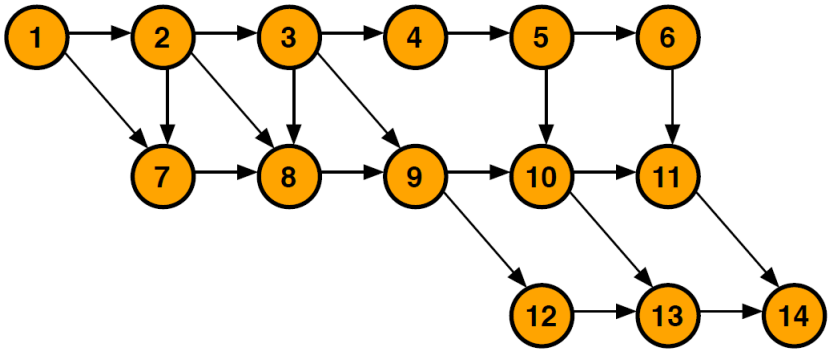

Figure 1: Thormund's Network

To calculate all of the potential paths from node 1 to 14, we will use the dynamic programming to fill our table. Below is the total number number of paths of Thormunds Network.

	0	1	2	3	4	5
0	18	15	9	3	2	1
1	X	3	3	3	2	1
2	X	X	X	1	1	1

By doing dynamic programming approach there is unique we know there are 18 unique paths available from 3 by 6 table from node 1 to node 14. We can see we can compute the number of paths from node 1 to node 14 by going through each node and adding the different paths from node 1 to node 14.
From 1 to 14:

$$(1 \longmapsto 2)(2 \longrightarrow 14) + (1 \searrow 7)(2 \longrightarrow 14)$$

$$(1) * (15) + (1) * (3) = 18$$

3. (20 points) As a new event in the Triwizard Tournament, students must compete in pairs in the following game. An even number n of magical items are laid out in a row, with the i-th item having a value of $v(i)$ in Knuts for each $i = 1, 2, \ldots, n$. The players alternate turns, and on each turn, the player may choose either the first item or the last item remaining in the row, then remove it from the row and add it to their pile. The player with the highest-valued pile at the end wins. (The value of a player's pile at the end is simply the sum of the value of the items in the pile.) We will use dynamic programming to help us analyze this game.

   (a) For a given vector v of values, let $F(i, j)$ denote the maximum value that the first player can definitely achieve using the values $v[i], \ldots, v[j]$—this means a *tight*, *exact* upper bound, an amount which the first player cannot beat, but that the first player can in fact achieve if he/she plays optimally. Write down a recurrence relation for $F(i, j)$ and prove that it is correct. Be sure to include a base case! You can assume that this problem has optimal substructure.

   (b) Based on your recurrence from part (a), describe a dynamic programming table and the order in which it should be filled in.

   (c) Write pseudo-code for the dynamic programming solution you described in part (b) and analyze its run-time.

(a):

S'pose the problem has an optimal sub structure, and Considering our statements above. The first player either will choose vi. The second player can choose vi+1 or vj. It is up to the opponent to leave the user with a minimum value. So we have 2 equations in our Recurrence Relation Below:

F(i, j) = MAX[vi+min(F(i+2, j), F(i+2, j-1))
= vj+min(F(i+1, j-1), F(i, j-2))

Base Case:

F(i, j) = vi if j == 1
F(i, j) = MAXvi if j = i + 1

(b):

Our dynamic programming table and the order in which it should be filled in Diagonally. This is because our recurrence pattern will result in the following sequence:

F(i+1, j-1) $\longrightarrow F(i, j-2) \longrightarrow F(i+1, j)$

As we see from the values above we have a diagonal pattern.

(c):

```
Code implemented in C++

// Here we create a table to store solutions of sub problems
//and then we Initialize them to zero diagonally




int Array(int* arr, int n)
    {
        int table[n][n];
        for(int i=0;i<n;++i){
            for(int j=0;j<n;++j){
                table[i][j]=0;
    }
}


// Here we have our conditions to check for 0

for (int gap = 0; gap < n; ++gap) {
for (int i = 0, j = gap; j < n; ++i, ++j) {




// From above we know x is value of F(i+2, j), y is F(i+1, j-1)
    int x = ((i + 2) <= j) ? table[i + 2][j] : 0;
    int y = ((i + 1) <= (j - 1)) ? table[i + 1][j - 1] : 0;
    int z = (i <= (j - 2)) ? table[i][j - 2] : 0;

    table[i][j] = max(arr[i] + min(x, y), arr[j] + min(y, z));
        }
    }

//  printed  statement:
for(int i=0;i<n;++i){
```

```
    for(int j=0;j<n;++j){
        cout<<table[i][j]<<" ";
                        }
        cout<<endl;
        }
    return table[0][n - 1];
}




int main(){
int arr1[] = { 1,3,7,2 };

int n = sizeof(arr1) / sizeof(arr1[0]);

printf("%d\n", CoinGame(arr1, n));


return 0;
}
```