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).*

alignStrings(x,y)- Firstly I made the matrix one bigger than the length x and y as it's needed for basecases. Then fill up the base cases increasing by 1. Then you loop trhough it filling it up using the cost function (Sub, Indel, Swap and no-op). it check no-op first and if its true then doesn cost(i-1,j-1) + 0 if not then checks all other operation and picks the min of them.

extractAlignment(S,x,y)- This does something similar to alignString but instead of doing cost it adds the operation that gets the min and adds it to the matrix represented by a symbol.

`commonSubstrings(x,L,a)`-

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

It would be $O(|x| * |y|)$ as common Substrings is using the matrix and not using it inside another loop so its 2(—x—*—y—) =¿ —x—*—y—.

(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.

(Answer)

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.

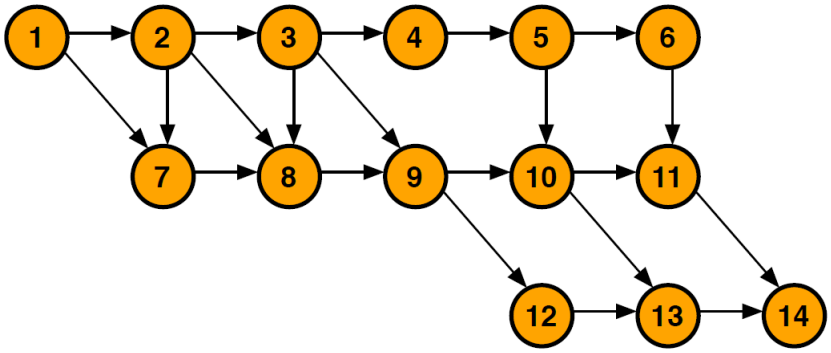

Figure 1: Thormund's Network

Node14 = 1
Node13 = Node14
Node12 = Node13
Node11 = Node14
Node 10 = Node11 + Node13
Node9 = Node10+Node13
Node8 = Node9
Node7 = Node8
Node6 = Node11
Node5 = Node6 + Node10
Node4 = Node5
Node3 = Node4 + Node9 + Node8
Node2 = Node3 + Node8 + Node7
Node1 = Node2 + Node7

So the Number of paths from Node1 to Node14 is 18

N14	N13	N12	N11	N10	N9	N8	N7	N6	N5	N4	N3	N2	N1
1	1	1	1	2	3	3	3	1	3	3	9	15	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.*

   The First player has 2 option: choose v(i) or v(j). Depending on your first choosing the second player can choose between (i+1 or v) or (i or v-1). So from this we can create if first player chooses v(i) then $v(i) + min(F(i+2, j), F(i+1, j-1))$ otherwise if he chooses v(j) then $v(j) + min(F(i, j-2), F(i+1, j-1))$ and from this we can create:

```
F(i,j) =  MAX(v(i) + min(F(i+2,j) ,  F(i+1,j-1) )
                    ,v(j) + min(F(i,j-2) , F(i+1 , j-1) ))

Base case:
when i = 0 and v=1 (smallest amount items allowed: 2) or more specifically
F(i,j) = v(i) if i==j
F(i,j) = v(j) if i==j
F(i,j) = MAX(v(i),v(j)) if i=j-1
```

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

   It would start filling it in diagonaly as thats the items (F(1,1) , F(2,2)...etc) then using the recursive relationg you start filling the rest (F(i,i+1) until you hit F(i,v) for every i and) :

Usingf items value (2,1,7,3) I filled a table with the order they were places in Skipping the zeros as they can be initialize and skipped in program.

Null	I1	I2	I2	I3
I1	2(1st)	2(5th)	8(8th)	9(10th)
I2	0	1(2nd)	7(6th)	4(9th)
I3	0	0	7(3rd)	7(7th)
I4	0	0	0	3(4th)

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

```
MaxVal(V):
n=V.length
inititialize table[n][n] = 0 in all slots
for gap = 0 to n-1
    for i=0,j=gap to n-1
        if i+2 <= j
            x=table[i+2][j]
        if i+1 <= j-1
            x=table [i+1][j-1]
        if i <= j-2
            x=table[i][j-2]
        table[i][j]= max(V[i] + min(x,y) , V[j] + min(y,z))
```