1.a.i. The function alignStrings works by taking in the two strings to be aligned as inputs. First the cost matrix S is initialized with a size that is length of first input string plus one by length of second input string plus one because we need to insert the "blank" space. The cost matrix is filled with large numbers so the min cost will always be lower than the initialized values. Then the first row and column are filled with indel operations calculating the cost to align the strings with a blank space. Next the rest of the matrix is filled in. First, my algorithm runs through each reaming space in the cost matrix S one by one. It checks if i and j are greater than 2 so the cost of S[i-2][j-2] + transposition can be computed. If not the variable for this value will be set to a large number. Then the remaining 3 costs are computed looking up, left, and to the corner value (S[i-1][j-1]). After, all the values are compared and the minimum of all 4 values is set to be the value of the current element in the cost matrix S. This repeats until the cost matrix is complete. This process uses two outside functions, sub and trans, which I have coded. Sub calculates the cost of a substitution by taking in two characters and comparing them. If they are equal the sub cost is 0, but if they are different the sub cost is 1. Trans calculates the cost of transposing two substrings. This transpose is used when calculating the cost from S[i-2][j-2]. It calculates the cost of the initial swap and the possible costs of a 0-sub cost, 10-sub cost, or 20-sub cost. These sub costs all depending on if certain characters are equal or different from one another.

ii. The function extractAlignment creates an array that contains the optimal costs that led to the entire optimal cost of aligning strings x and y. I wrote two separate functions for this function, determineOptimalOp and updateIndicies. So this main function iterates from S[i][j] to S[0][0], finding the optimal costs along the way. i is set to length of x and j is set to length of y. First a is appended with the optimal op from which the current i and j element came from. Then the indices of i and j are updated from where the optimal cost came from. In the determineOptimalOp function all 4 cost operations are calculated, but depending on the values of i and j, only some operations will be calculated since for example, S[i-1][j] + cost(indel) cannot be computed if i = 0. Once the costs, that can be computed, have been computed the optimal cost is returned. The extractAlignment function then updates array a. The function updateIndices looks at the last element added to array a, which is the last optimal cost that built the cost at S[i][j], and moves indices i and j according to where it is likely to have come from. If the current value of a is 0 then it was a no-op and the indices are set to i-1, j-1. If the current value of a is one then it either came from above or left, so conditions on the indices and values of i and j will determine whether indices are updated to go up or left. Then if the current value is bigger than 1 it had to have been a transposition and the indices are set to i-2, j-2. Then extractAlignment returns the array a.

iii. The function commonSubstrings first creates an empty array that will hold all the substrings of x that align with substrings of y via a run of no-ops. Then it runs a while loop with the condition that the index that runs through the array of optimal costs—a—is less than the (length of a) minus one. It checks if the current value of a is 0, if it is a no-op, and if it is it enters a while loop that keeps iterating through a counting how many no-ops are consecutive. Once a value other than 0 is encountered in array a that inner while loop breaks and the length of that chain of no-ops is checked against the input size L. If it is greater than or equal to L then that substring of x is pulled out and appended to the substrings array. The count is then set back to 0 so that the next string of consecutive no-ops can be found on the next pass through the while loop. After the entire array a has been sifted through and the substrings from x that have been found, the function just returns those substrings.

b. Let n = length of x and m = length of y. The running time of the call commonSubstrings(x, L, extractAlignment(alignStrings(x, y), x, y)) would be Θ(nm). The running time of alignStrings is the largest of the three functions because it visits every value of the cost matrix S which is of size (len(x)+1)*(len(y)+1). The running time of extractAlignment is Θ(max(n, m)) because it either goes through every letter that is to be aligned or it only goes through half of the letters by grouping them in strings then aligning them. Either way its running time is still Θ(max(n, m)). The running time of commonSubstrings is Θ(len(a)), where a is the matrix of optimal costs and is smaller than both n and m. This is because commonSubstrings goes through every value in a. Summing these running times up we get nm + max(n, m) + len(a). Therefore, the running time, of the call above, is Θ(nm).

d. [' are mostly h', 's and more', 'te more than ', 'lanned or un', 'ar-reaching and long', ' are expected to have ', ' operating at mature levels', 'ted to con', 'wth approa', 'n the united sta', 'reating more than ', ', d.c. -- ', 'ent program washingto', 'reating invest', ' president trump congratulates ex', 'the white house office of the press secretary for immediate release march 0']

2. To count the number of paths from nodes 1 to 14 we will use a recursive process. We will count the number of paths from the current node to 14, starting at 14. There are 0 paths to 14 from 14, so we move on to 13. 13 has 1 path to 14. Then we move to 12. 12 has 1 path to 13, so well add the total number of paths from 13 to 14 to paths from 12 to 14 and that will be the total paths from the current node, 12. We will repeat this process until we reach node 1 where the paths from 2 to 14 and 7 to 14 will be summed to give the total paths from 1 to 14. We can see this process in the table below.

Node	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Paths	0	1	1	1	2	3	3	3	1	3	3	9	15	18

So, the total number of paths from node 1 to node 14 is 18 paths.

3.a.i. The array L is filled by recursively calling MemLuc(n) until the base cases are returned (i.e. MemLuc(1) or MemLuc(0) is called). Once those base cases are returned, the Lucas Numbers up to n are computed one by one and stored in the array L.
ii. Base case MemLuc(2) = 3
Assume that MemLuc works for some number k where 2 < k < n.
Induction step: Prove MemLuc works for k+1.
Since we have assumed that MemLuc works for some number k, that means that MemLuc(k) = MemLuc(k-1) + MemLuc(k-2) where MemLuc(k-1) and MemLuc(k-2) have already been calculated. So MemLuc(k+1) = MemLuc(k) + MemLuc(k-1). This holds true because we already have the values of MemLuc(k) and MemLuc(k-1), making MemLuc(k+1) the correct Lucas number for k+1.

b. The time and space usage of DynLuc(n) are both Θ(n). For the time complexity, the algorithm assigns every element of a length-n array to its respective Lucas number using a for loop. So, the time complexity is Θ(n). The space usage is also Θ(n) because it is keeping track of a length-n array that stores all the Lucas numbers up to n.

c. The bug in Hermione’s code is in her return statement. It should read return c not return a. Returning c returns the Lucas number for n where a returned the Lucas number for n-2. The time complexity of FasterLuc is Θ(n) because it loops through every number from 2 to n. The space complexity of FasterLuc is Θ(1) because at each iteration 3 separate variables are storing a constant integer. This algorithm does not store every Lucas number up to n it only stores the current Lucas number and the two previous to that Lucas number.