Pset0

Problem Set 0

Problem 0: Collaboration Policy (10%)

Signed on HuskyCT

Problem 1: Introductions (10%)

I'm a double major in computer science and finance framiliar with python, Java, bash, and the linux kernel. I work during the Semester as a software engineering intern at Cigna Healthcare, as well as at my own side projects and businesses. I have a mind for business, and my future goals include software management, entreprenuership of my own company, and promoting the freedom of software though building and funding open source projects. I have stong opinions about the future of software being open and accessible, and the societal changes promoting open source could lead to.

Problem 2: Preliminaries (10%)

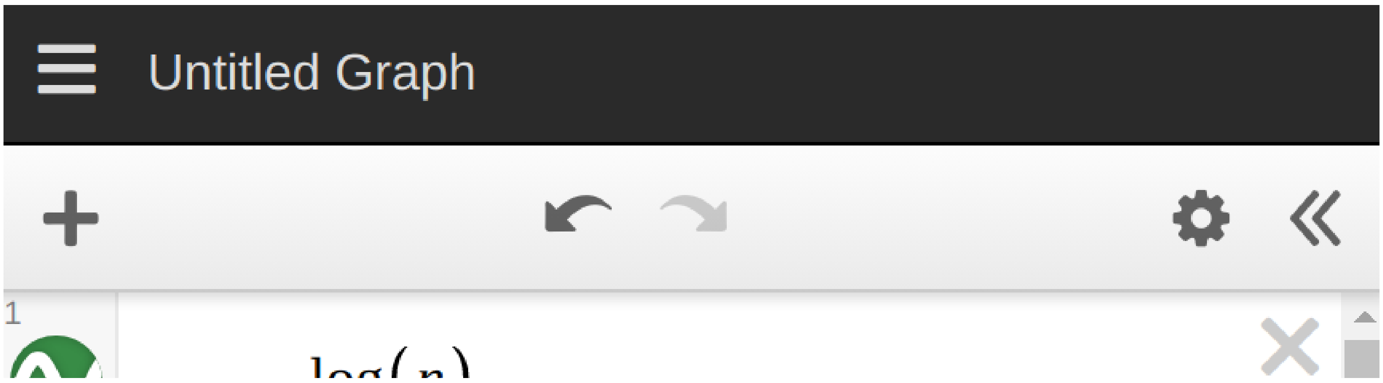
1. How many Permutations are there of the set of n numbers? $\binom{n}{n}$
2. How many Subsets of three numbers are there from a set of n numbers? $\frac{n!}{(n-3)!}$
3. How many subsets of S are there when |S| = n? 2^n

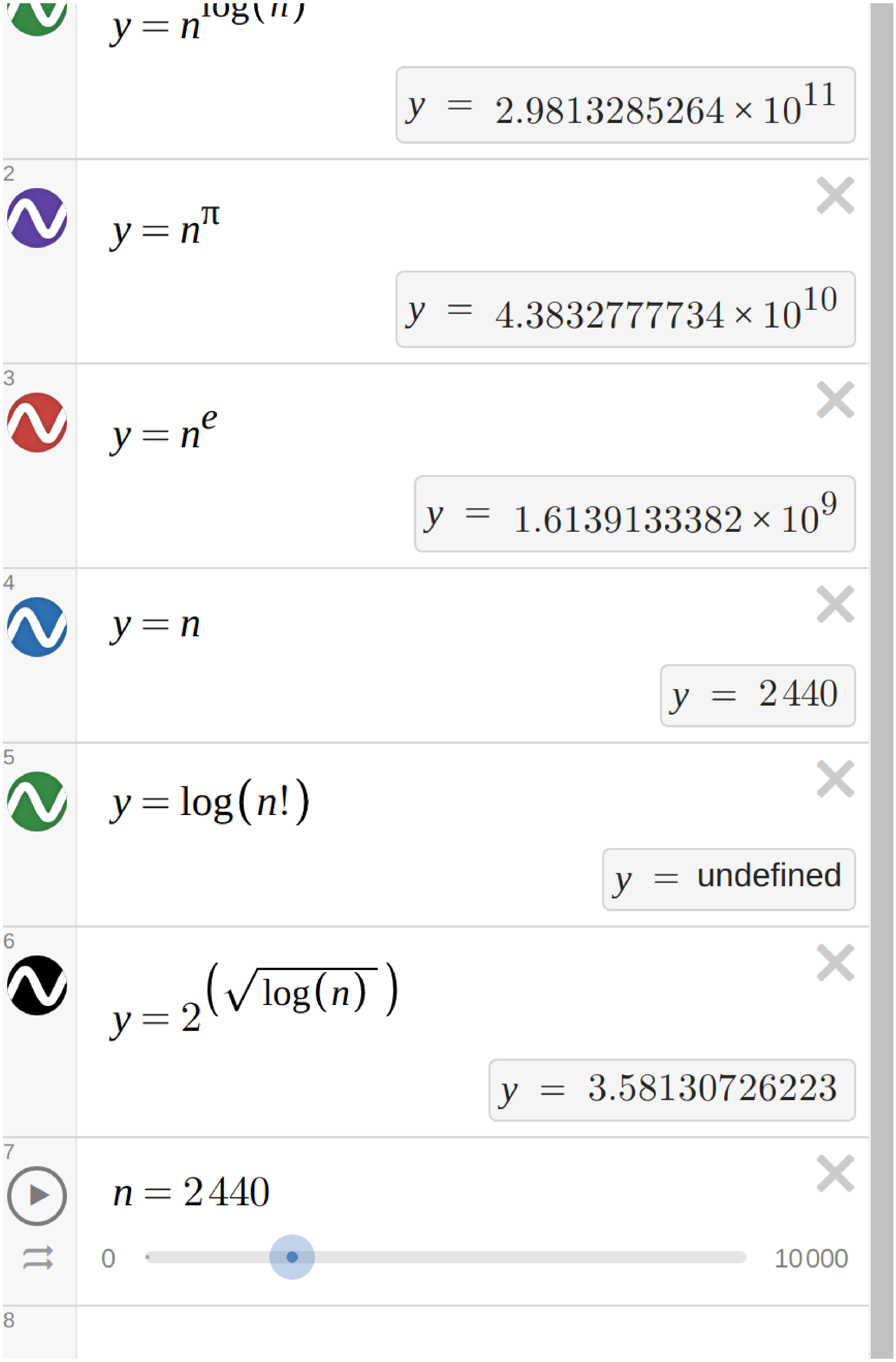
Problem 3: Complexity (20%)

Question 1:

1. $A = n^c, B = c^n : A = \Omega(B)$
2. $A = n^{lg(c)}, B = c^{lg(n)} : A = \Theta(B)$
3. $A = lg(n!), B = lg(n^n) : A = \Theta(B)$
4. $A = 3^{3^n}, B = 3^{n^2} : A = O(B)$

Question 2:
Therefore;





$y = n^{\log(n)}$

$y = 2.9813285264 \times 10^{11}$

2

$y = n^{\pi}$

$y = 4.3832777734 \times 10^{10}$

3

$y = n^{e}$

$y = 1.6139133382 \times 10^{9}$

4

$y = n$

$y = 2440$

5

$y = \log(n!)$

$y =$ undefined

6

$y = 2^{\left(\sqrt{\log(n)}\right)}$

$y = 3.58130726223$

7

$n = 2440$

0 10000

8

powered by

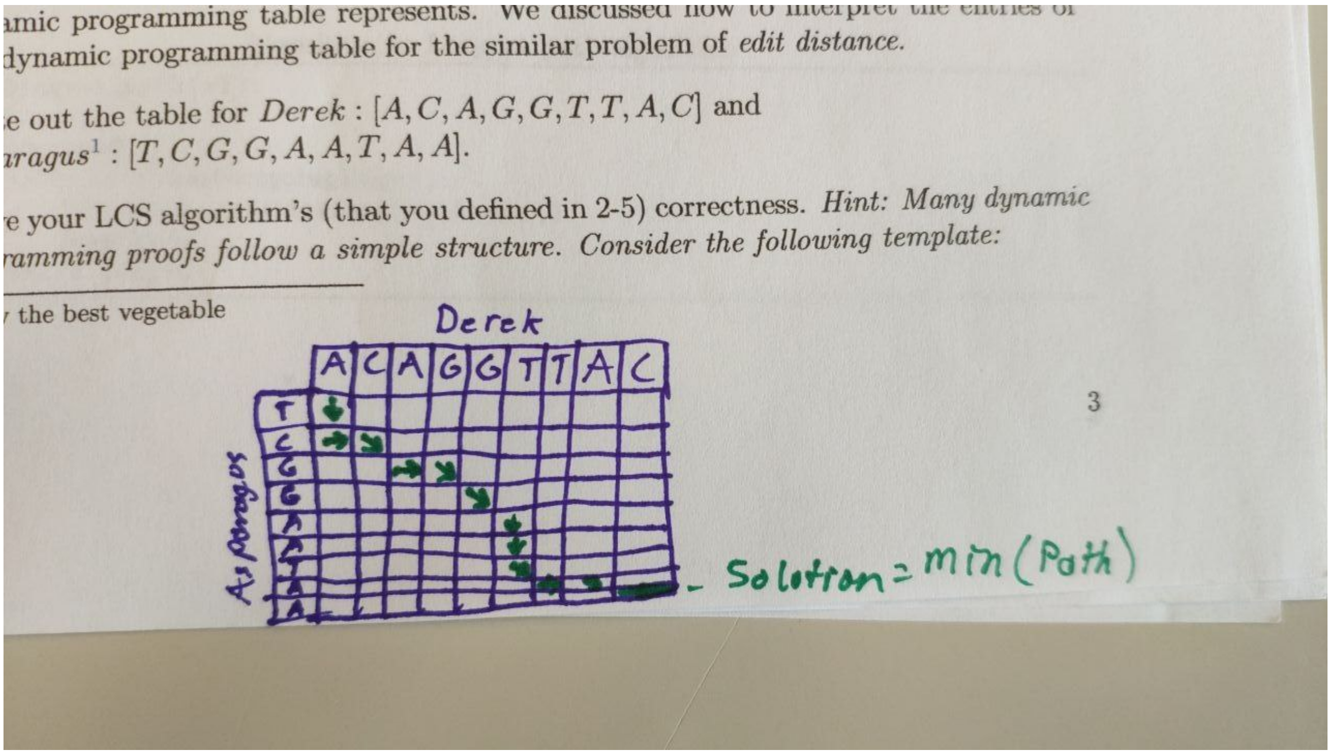
In descending order;

1. $n^{log(n)}$
2. n^{π}
3. n^{e}
4. n
5. $log(n!)$
6. $2^{\sqrt{l}og(n)}$

Problem 4: Vinder (30%)

1. A "Brute Force" method of solving this problem could be breaking the template strand of DNA into all of its respective subsets where template = T, meaning the template would have $\binom{t}{t}$ possibilities. Then, one could take the strands of all vegetables and break them into their respective subets where vegedna = v, meaning the vegedna would have $\binom{v}{v}$ possibilities. This solution would be obviously impractical as its runtime would be nearly infinite as t and v scaled up. We do however know it will find the correct solution becuse we know the correct solution is a subset of both t and v which match, and since we calculate all the possible subsets of the templates from t and v, and then compare them, we know at some point we will find the optimal solution (on magic hardware).
2. The LCS can be broken into subproblems by only comparing the first section of the prefix, finding an optimal solution that matches that prefix, and extending it by 1 to solve the next subproblem (assuming each subproblem solved before it was the "optimal solution"). Since the subproblem here is comparing two values, we know that if two values equal each other, we can move to the next section of our sequence, whereas if they do not, we terminate and record the length and location of the sequence. If we solve enough of these subproblems, the optimal solution will be the one with the longest length of matching sequences between our template sequence and the vegetable sequences.
   $E(i, j) = max(cases...)$
   1. $(E(i-1, j-1) + 1$
   2. $E(i-1, j)$
   3. $E(i, j-1)$
3. Converting this to a dynamic programming problem would be similar to our word-completion problem. The size of the table would be the length of the longest string +1. This could be done by lining one string on the x axis, and another on the y axis, then comparing the two as you move down them. (similar to in class either we can gap one, gap the other, or match corresponding to a value)
4. Each entry could either compare and have them be equal, in which it moves on to the next value of the template string T, on our table that would be equivalent to moving horizontally and vertically if

our templates were on the X-axis and Veggie string V on the Y-axis. If we then can either increment i or j to create a gap on either end which visually on our table is a horizontal move (to the right) if incrementing our template string, and vertically (down) if incrementing our Veggie String. If we take an oblique path, we'd measure the distance as 0 whereas other would have their distances measured at 1. Our shortest path would be the sequence of "best matches" between the veggie string and the template string.

5. 

amic programming table represents. We discussed how to interpret the entries of dynamic programming table for the similar problem of *edit distance.*

e out the table for *Derek* : $[A, C, A, G, G, T, T, A, C]$ and *ragus*[1] : $[T, C, G, G, A, A, T, A, A]$.

e your LCS algorithm's (that you defined in 2-5) correctness. *Hint: Many dynamic ramming proofs follow a simple structure. Consider the following template:*

the best vegetable

3

Where the solution is all of the oblique arrows.

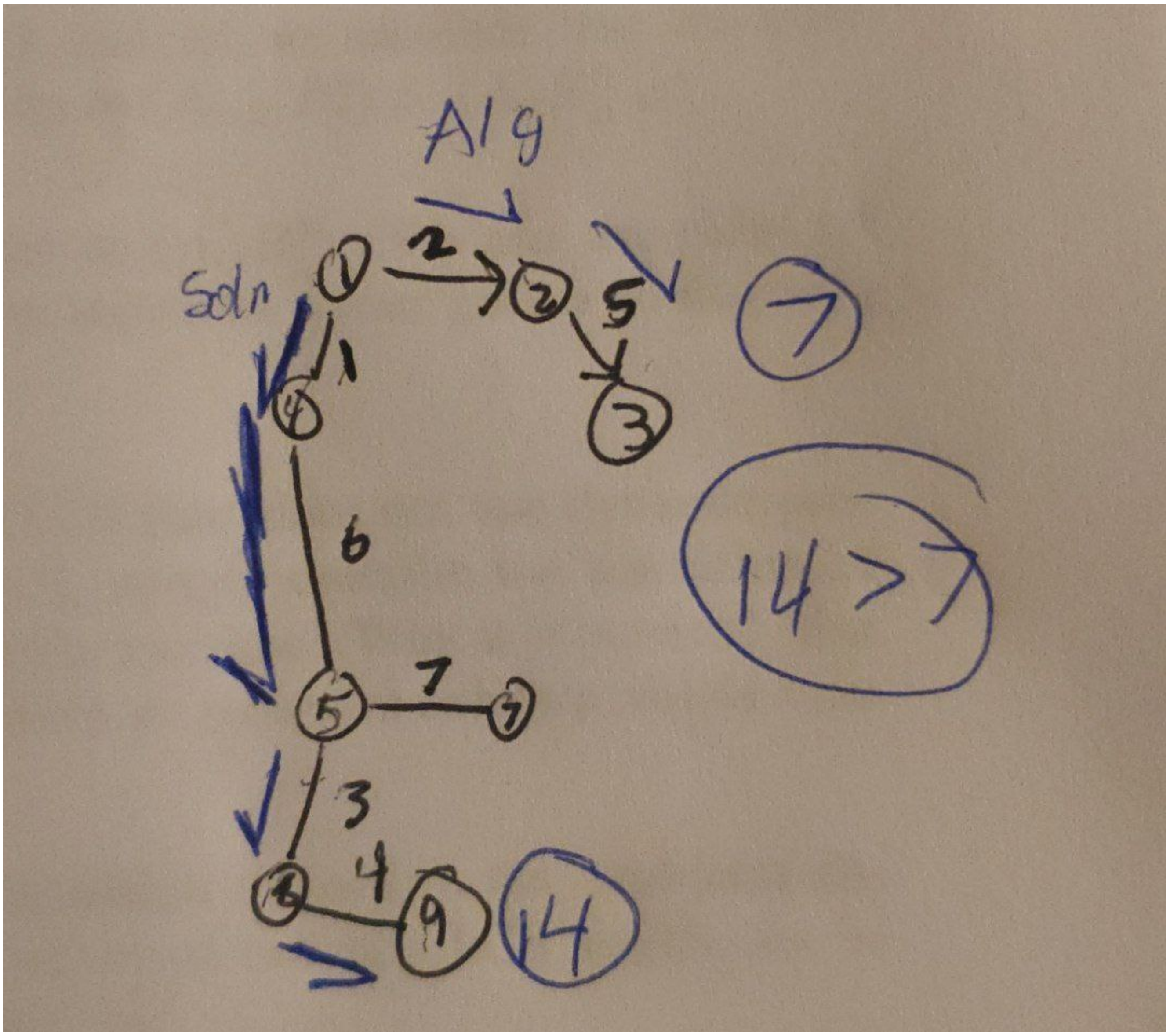
6. a)I will show that, given the previous subproblems were solved correctly, this algorithm will be able to find the longest common string between two strands of DNA. This is done by computing the following formula $E(i, j) = max(cases...)$
   1. $(E(i-1, j-1) + 1$
   2. $E(i-1, j)$
   3. $E(i, j-1)$

   which finds the longest prefix, and the max of the prefixes generated will contain the LCS.
   b)The induction hypothesis assumes that the shortest path to the i=i i'th and j=j j'th (from i=0, j=0 to i=i, j=j), there exists an LCS which is the best matched substring between the template DNA and the Veggie DNA. The Base Case is that the Veggie DNA is equal to the Template DNA (you are a vegetable). Either the letters match, you move forward on the template string, or you move forward on the Veggie String.If all of the previous calculations were correct, and you have the shortest path from the start to where you are currently, then by taking the maximum sized step (the one which adds 1 when they match), you will compute the longest prefix. This can be used to find the LCS as the longest prefix for the entire set should be the longest common string.
   c) To get the optimal solution, you must either match characters, which adds a length of one to a string, or gap on your template string or veggie string. The subproblem is finding the match which means increasing your string length by one. If every previous subproblem was solved

correctly, you should have the longest string possible thus far, and if you pick the max value of this set of values, will end up with the LCS.

Problem 5: Longest Path Problem

1. Alg1 does not correctly solve the problem because 'iter.getEdgeSmallestIndex()' is returning the smallest index, rather than the longest edge with the lowest index (taking the $min_j\{(iter, v_j) \in E$ } returns the edge with the lowest node value rather than the highest weight. This means the directed graph could potnetitally lock iteslf into a short path which ends with a dead end.



2. The problem with their implementation is they didn't consider the distances. First, in order to find the longest path in the graph, we must find the longest path from our start node to the next node. To do this, we would likely need to first sort our graph in topological order to make the processing simpler. Now you need to go to the max path's distance. $max\{dist(v_i)$+1, $dist(v_j)$+1}
This should find the longest path from the vertex you are on, to the next edge. Repeated, this algorithm would be written:
for vertex v in set V in linearized order:
longestdist(j) = $max_{(i,j)} \in E\{dist(i) + 1\}$