COMPSCI 311: Introduction to Algorithms **Spring 2022**

Homework 4

Released 3/24/2022 Due 4/6/2022 11:59pm in Gradescope

Name: Brendan Cadogan Collaborators: Ben Tufano, Nate Courchesne

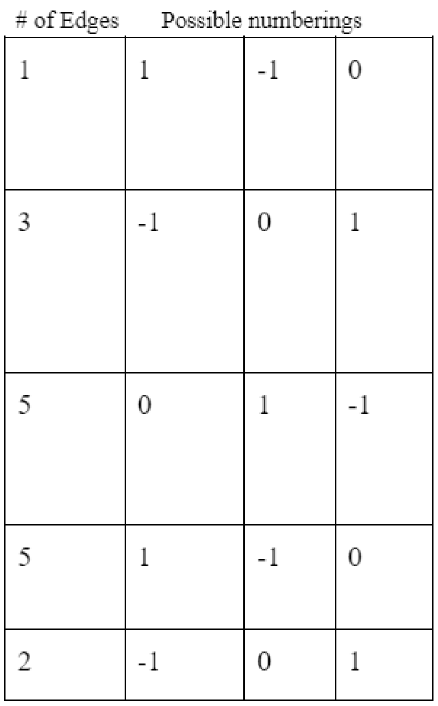
Instructions. You may work in groups, but you must write solutions yourself. List collaborators on your submission. You are allowed to have at most 4 collaborators. Also list any sources of help (including online sources) other than the textbook and course staff.

If you are asked to give an algorithm, please provide: (a) the pseudocode or precise description in words of the algorithm, (b) an explanation of the intuition for the algorithm, (c) a proof of correctness, (d) the running time of your algorithm and (e) justification for your running time analysis.

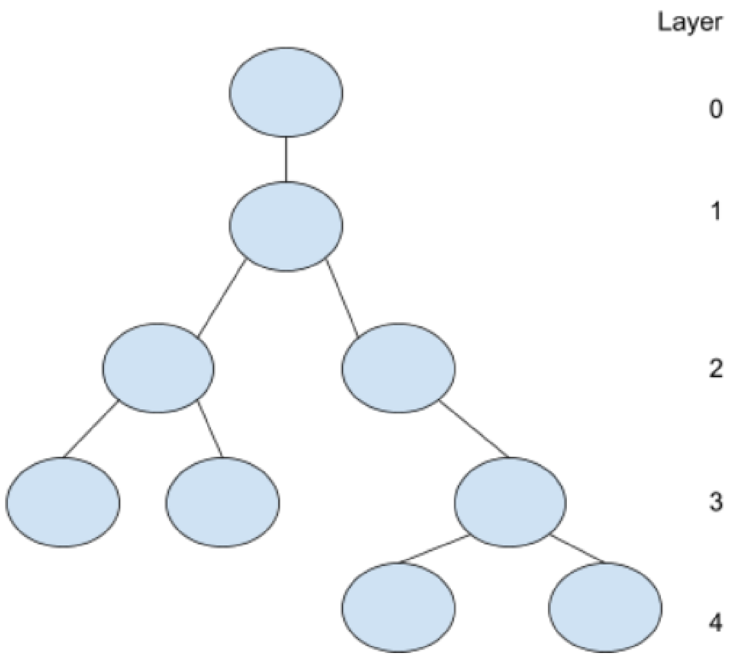
Submissions. Please submit a PDF file. You may submit a scanned handwritten document, but a typed submission is preferred. Please assign pages to questions in Gradescope.

1. **(20 points) Labeled Graph**
You are given a connected acyclic graph $G = (V, E)$, with each node of degree at most 3. Find a labeling of its nodes with values in $\{-1, 0, 1\}$ such that (1) each edge (u, v) has endpoints with different labels, $\ell(u) \neq \ell(v)$, (2) for nodes with several neighbors, not all have the same label, and (3) $\sum_{(u,v)\in E} |\ell(u) - \ell(v)|$ is maximal.

A. The brute force solution consists of taking our n nodes and trying all three possible combinations on it which would have a bad runtime of $O(3^n)$. To optimize this, we will instead try all 3 possibilities per layer instead of per node. Since we know the graph is a binary tree due to the fact that each node has at most a degree of 3 and it is acyclic, we can find a leaf node to start our algorithm from. We need to start from a leaf node because if we start from a node with a degree of three or of two as layer 0, our algorithm will give an improper solution due to constraint number 2. To get to a node with a degree of 1, we will iterate through all the edges and add 1 to the degree of the nodes we discover and take a node that has a degree of 1. We can keep track of nodes that have degree of by pushing back nodes with a degree of 2 or 3 to the end of the list. Next we will run a BFS search from one of the leaf nodes which will now be our root node. The BFS will label our layers, and at the same time, will give us the amount of edges the nodes have in that layer. We will then multiple the amount of edges for each layer by $1, |-1|$, and 0 and then add up the results three times using 3 patterns for the multiplication part. The patterns are 1,-1,0,1,-1,0,..., then -1,0,1,-1,0,1,..., and then 0,1,-1,0,1,-1,...., all three of which are represented in the graphic below. We will then take the biggest sum of the edges per layer multiplied by the patterns and return that node ordering.



# of Edges	Possible numberings		
1	1	-1	0
3	-1	0	1
5	0	1	-1
5	1	-1	0
2	-1	0	1



B. Like most dynamic problems, we wrote out the brute force solution and tried to notice something we could optimize.

C. To prove our algorithm, we need to prove that one of them will always give us a maximal solution. The first thing I would like to mention is that there are other ways to label the graph outside of the three patterns we used, but our ways always produce one of the maximal solutions. The reason our pattern goes 1,-1,0 is that by our rules, we can't have the same number at both ends of an edge, so if we label one layer 1, then another layer -1, the next layer must be 0 and not 1, and then the next must be 1 not -1. Another thing is that in our patterns, we can switch -1 and 1, so our patterns could also be -1,1,0,..., 1,0,-1,..., and 0,-1,1,... because we take the absolute vale of $l(u) - l(v)$ which doesn't change based off of where 1 and -1 are in that equation. Now that we know why we have the patterns we have, we need to prove that they produce a maximal solution. They do because at least one of our patterns always has the maximal amount of edges with a 1 or -1. We want the maximal amount of edges with 1 or -1 at the end of them because that for every 1 or -1, that is another 1 to the sum.

D. The run time is $O(m+n)$.

E. The run time is $O(m+n)$ because we first we do an iteration over each edge to get the degrees of each node to know where we can start from. That part is $O(m)$ because we iterate over each edge once. Then we do a BFS to determine what nodes are in each layer and the amount of edges in each layer which is $O(m+n)$. We then do the layer multiplication and summation which are constant time, which means our final time is $O(m+n)$ because $O(m)$ is dominated.

2. **(20 points) Maximum Result**
You are given a list L of n positive reals; $v(i)$ is the value of the i^{th} list element. At each step, choose an index $1 \leq k < \text{length}(L)$ and replace the two values $v(k)$ and $v(k+1)$ with one value, $\sqrt[3]{v^2(k) + v^2(k+1)}$. Find the sequence of indices to choose so that after $n-1$ steps, the only remaining list element is maximal.

A. Our algorithm is an optimized version of the brute force algorithm. The brute force algorithm consists of combining every possible combination, then combining every possible combination of those, and on and on until you have every possible answer and then you return the sequence that gives you the maximal value at the end. The problem with this is that it has a run time of O((n-1)!) which is equal to O(n!), and we want to minimize this. To do this, we will calculate the maximal value, and then keep doing that until we have combined every number to one number. For example, if our array is [a,b,c,d,e], we will calculate ab, bc, cd, and de, and then choose the maximal of those. If the maximal is cd, our new array would be [a,b,(cd),e]. We take this new array, and do the same thing, and keep repeating these steps until we have one number left in the array. We then return the sequence of indices used to get there, something we will store in an array. The recurrence call for our algorithm is $OPT(i,j) = max_{i<k<j}((OPT(i,k)^2 + OPT(k+1,j)^2)^{\frac{1}{3}}$.

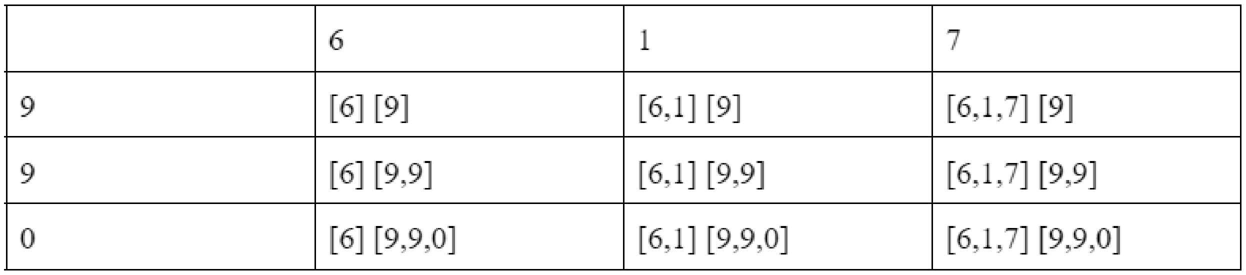
B. We took the brute force algorithm of calculating every possible combination of merging with run time of O(n!), but then we realized that to find the maximal solution, it meant finding the maximum solution at every instance.

C. To prove this, we need to prove that we choose the maximum at each instance and therefore we get a maximal solution. This is true because we try every possible combination, and then pick the maximal to combine and continue from there. D. The time complexity is $O(n^3)$

E. The time complexity is $O(n^3)$ because our recurrence is $T(n) = T(n-2) + 2$ which equals $\Theta(n^2)$. We then do this recurrence n times so $n^2 * n = n^3$ and therefore we have $O(n^3)$

3. **(20 points) Increasing Joint Subsequence**
Given two sequences of reals $A[1..m]$ and $B[1..n]$, construct the longest strictly increasing sequence from them: at each step, choose an element of A or B; thereafter, you may not choose an earlier element of that sequence.

A. This problem is a variation of the longest increasing subsequence problem we solved in discussion 7. The recurrence for that is $OPT(j) = max_{i<j:A[i]<A[j]}(i + OPT(i))$. We will solve the LIS n^2 times and return the biggest sequence we find.



	6	1	7
9	[6] [9]	[6,1] [9]	[6,1,7] [9]
9	[6] [9,9]	[6,1] [9,9]	[6,1,7] [9,9]
0	[6] [9,9,0]	[6,1] [9,9,0]	[6,1,7] [9,9,0]

The above graph shows us which values we merge sort and run LIS on.

```
for(i=0;i<m;i++)
  for(j=0;j<n;j++)
    LIS(merge(A[0 till i],b[0 till j])
```

As you can see, we take the first element in both arrays, merge them, and then run LIS on it. We merge them by choosing the shortest on the front, so if we have [6,1,7] and [5,2,4,7], we would have [5,2,4,6,1,7,7].Then we do that for the first two elements of one array and the first element in the other. We do that until we get to the nth element. We also do this for the other array, which means we run the LIS n^2 times. When we are done and the recurrence returns the max length, we still need to return the actual sequence. To do this, we'll store each sequence we find in an array and return the longest sequence when done.

B. We looked at the brute force algorithm and tried to optimize it. The brute force algorithm consists of trying every possible combination which would take $O(2^n)$. We also found the LIS problem in discussion 7 which helped us a lot.

C. The first thing I would like to mention is that we know the LIS works because we proved that in discussion 7. The next thing we need to prove is that our shortcut of the brute force always returns the longest increasing subsequence. It does because our method cuts out recalculations. This is because if we draw out the tree of the brute force, we'll notice repeats, but if we draw out the tree of our version, we get rid of the repeats and some impossible solutions.

D. The run time is $O(n^3 logn)$.

E. The run time is $O(n^3 logn)$ because we merge our sub arrays which is O(m+n), then do the LIS which is $O(nlogn)$, and do this $O(n^2)$ times. The $O(nlogn)$ dominates the O(m+n) so we get $O(n^3 logn)$.

4. **(20 points) Pebble Game**
In a game with three piles of pebbles, two players take turns, doing one of three moves: (1) remove two pebbles from one pile, (2) remove three pebbles each from two piles, (3) remove one pebble each from three piles. A player who cannot move loses. Give an algorithm that determines which player has a winning strategy, starting the game with three piles of $n \geq 1$ pebbles each, and what that strategy is.

A. Our algorithm consists of filling a 3D array with true or false values. The indexes of the array represent how many pebbles are in each pile, so i would be pile 1, j would be pile 2, and k would be pile 3. The true values represent that if it is your turn at that instance of the game, you have a winning strategy. If it is false, you don't have a winning strategy. Our algorithm will initialize a 3D array based off of the initial amount of pebbles. To calculate if the value at a certain index is true or false, we need to check the values of the indexes that would represent the possible moves from that index. Our base case will be the square from which all possible moves our out of bounds, and therefore we set them to false. For any index that is not our base case, if all of the possible moves from our value are true/out of bounds, we set our value to false, otherwise, we set our value to true. Now there are two main ways to calculate the values necessary for determining if our instances of the pebble game will have a winning strategy. The first is to starting from [0,0,0] and using three for loops to calculate every value while the second is a recursive call from our initial index so that we calculate only the values necessary. We'll use the iterative way, which will go from index to index calculating the true or false value based off of the previous turns until we get to our turn, in which we'll return true if there is a winning strategy for our player and false if there is not.

B. To solve this, we thought of it as a game with 2 problems and only 2 possible moves, and then when we found the solution for that, we just modified it to fit our 3 pile 3 move problem.

C. To prove this, we can use strong induction. Our base case is all of the possible combinations of pebbles in which there are no possible moves, that is when we have 0 pebbles in all piles, 0 pebbles in 2 piles and 1 pebble in the third, and then 1 pebble in 2 of the piles with 0 in the third. These all return false because we do not have a winning strategy because there is no possible move because all moves are out of bounds. Next by strong induction, we can assume that all values from [0,0,0] to [x,y,z] are either true or false, and if they are true, making a move from that index will either return an out of bounds or a false value. If our value at [x,y,z] is true, it means that in [x-1,y-1,z-1], [x-2,y,z], [x,y,z], [x,y,z-2], [x-3,y-3,z], [x-3,y,z-3], and [x,y-3,z-3] there is at least one false because by definition, true represents a winning strategy so the next move from that indexes must not have a way for the other player to win, and therefore be false. Now if [x,y,z] is false, that means that there are all true value/out of bounds in [x-1,y-1,z-1], [x-2,y,z], [x,y,z], [x,y,z-2], [x-3,y-3,z], [x-3,y,z-3], because if there were a false, it would mean when you moved there and it is now the other player's turn, they would not have a winning strategy, and therefore the square you were just on should have been true because you would have the winning strategy. Since the inductive step was proven, that means our algorithm is proven.

D. The run time of this is $O(n^3)$

E. The run time of this is $O(n^3)$ because we use three for loops that iterate through the 3D array, and each for loop iterates n times, so therefore the runtime is $O(n^3)$.

5. **(20 points) Mountain Climbing**
You are given an $N \times M$ grid. Each cell holds a value. Your aim is to go from cell $(0, 0)$ to cell $(N-1, M-1)$, collecting the values on your path. From each cell, you can only go right or down (increment one coordinate). What is the maximum sum of values you can get from your path?
(10 points) Extra credit: Submit an implementation in Java or Python to the Gradescope autograder. Input/output specifications provided on Moodle.

A. We chose to use an algorithm similar to Bellman-Ford, except instead of finding the shortest path, we want to find the longest path. The first thing we want to do is create our directed graph in which each point has an edge going down and going to the right as long as there are values in that direction. The edge weights for these edges will be the values in the array. We also want to have a max value that we return, which will be initialized to hold the value at [N-1,M-1] because in the way we initialize our graph, it would be impossible to collect, and we always collect it anyways so we should do it at the beginning. The second thing we want to do is implement Bellman-Ford but with preference to longer edge weights. To do this, all we do is take our Bellman-Ford-Moore implementation from the slides and change two things. The first is that we set all nodes to -∞ instead of ∞, and the second is that we change in the if statement the > sign to a < sign. What this does is it gives us the longest route instead of the shortest, which is what we want. We want to keep track of the longest path and then return that when we finish.

```
construct a DAG from our N x M grid
set d[t] = 0 and d[v] = $-\infty$ for all v $\neq t$
set succ[v] = null for all v
mark t as updated
for i = 1 to n − 1 do
    for all nodes w updated in previous pass do
        for all edges v $\rightarrow$ w do
            if d[v] < c[v, w] + d[w] then
                d[v] = c[v, w] + d[w]
                succ[v] = w
                mark v as updated
            if no d[v] updated then
                stop
return max
```

B. Our intuition was that since there were negative edges, we couldn't used Dijkstra's because it doesn't handle negative edge weights. Then we thought that since the graph was a DAG, Dijkstra's would still work, but then looking back on the slides we realized this was not the case, so we settled for Bellman-Ford

C. The first thing we should mention is that this is a longest path problem on a DAG. To solve this, we would normally use a modified version of Dijkstra's, but there are negative weights so we know we need a modified Bellman-Ford algorithm. Because it is a DAG, we know that there will always be a possible solution, and that we can't get back to a node once we left it. This means that all we need to prove is that our algorithm finds the longest path, instead of the shortest path. We know that in the Bellman-Ford-Moore, we set the distance to all nodes to infinity, and replace the distance to get to that node only if our new distance is shorter, which gives us the shortest path. To instead get the longest path, we need to instead initially set the distance to every node to negative infinity and replace it when we find a distance that is greater, so that we find the greatest instead of shortest path. This works because of our new if statement, when we reach an undiscovered node we always take it, and when we reach a discovered one we only take it if the new value would be greater.

D. The run time is O(mn).

E. The run time is O(mn) because our algorithm is the Bellman-Ford-Moore implementation of a connected graph except three things. The first is that we needed to construct our graph, which is O(m). Secondly instead of setting our nodes to infinity, we set them to negative infinity. The third change is that we swap the greater than sign to a less than sign in our if statement.The second and third changes are O(mn). All three of these changes do not change the run time, because O(m) is dominated by O(mn) so we can say that it is still O(mn).

Attempt to make all of your algorithms efficient, and provide all elements listed in the instructions.