Please enter your name and uID below.

Name: Devin White

uID: u0775759

Collaborators, if any, and how you collaborated:

Submission notes

- Due at 11:59 pm (midnight) on Thursday, Nov 3rd.
- Solutions must be typeset using one of the template files. For each problem, your answer must fit in the space provided (e.g. not spill onto the next page) *without* space-saving tricks like font/margin/line spacing changes.
- Upload a PDF version of your completed problem set to Gradescope.
- Teaching staff reserve the right to request original source/tex files during the grading process, so please retain these until an assignment has been returned.
- Please remember that for this problem set, you are allowed to collaborate in detail with your peers, as long as you cite them. However, you must write up your own solution, alone, from memory. If you do collaborate with other students in this way, you must identify the students and describe the nature of the collaboration. You are not allowed to create a group solution, and all work that you hand in must be written in your own words. Do not base your solution on any other written solution, regardless of the source.

1. (Artisanal Cooking)

   NOTE: When referring to "numbers" it's referring to longs to prevent overflow
   The algorithm is structured in the following way: Parse the initial line into 2 long values, we'll call N and M, where N represents the total amount of food desired, and M represents the total number of dependencies, which will be used for parsing inputs later. Secondly, construct a (Global) new Dictionary of Dictionaries we'll call cookbook, as well three (Global) Arrays of Size N, one representing the foods called food, one representing whether the vertices are vistited or not, which we can call visited, and one representing the total amount of ingredients needed, we can call IngredientCount. The Food Array will be used to store the number of foods desired(the second line of input) and cookbook will hold our graph, with the outer dictionary keys representing a vertex, and the inner value, which is another dictionary representing a vertex and the number of that required to make the food that is the key in the outer dictionary(inner key=connected vertex, inner value = number required to make) this is Structured as $< Key, < Key, Value >>$ *or* $< FoodtoMake < FoodMaking, NumToMakeOuter >>$
   With our data structures now decided, we then begin parsing the rest of the input. For the second line, we iterate From i = 0 to N(ends at $N - 1$) and map the values in line 2 directly to the indexes in our food array as longs. After this, we parse the inputs of the dependencies into our dictionary of dictionaries(longs) with the outer key being the middle of the line(the food to make) the key of the "value" representing that food that makes the outer food, and the value of the value representing the number required.
   With our graph built, we can then begin the process of finding the solution to our problem. The problem will be solved by performing a depth first search/Topological sort, we do NOT need to store the topological order, we merely need to perform the necessary calculations as we are performing this DFS/topological sort.
   We do this by calling a method which will iterated from i = 0 to N, checks if the vertex is visited, and if it isn't, calls another method we will call DepthFirstSearch which will DFS by using recursion. In the recursive method, we begin by marking the vertex as visited(visited[i]), after this, we go through the keys of the inner dictionary mapped to our current vertex key, E.G $< currentVertex < ConnectedVertex, NumRequred >>$ and call depthFirstSearch on ConnectedVertex. This repeats for each connected vertex, until we get to the vertex at the "bottom" of a given path. When we leave this recursive loop, we perform out calculations. Which we will store in another array of size N we can call IngredientCount
   The calculations are formed as so:
   If the food is in the Dictionary(as an outer key) but has no values, the total amount of ingredients of that food = food[N]. Otherwise, we go through each of the inner keys of that vertex in the inner dictionary, take the values of the that inner dictionary with the associated key and peform the following calculation: $ingredientCount[outerVertex] += ingredientCount[innerVertex] * numberRequiredToMakeOuter$. After going through all the inner Vertexes, we then add $Food[outervertex]toingredientCount[outerVertex]$. This is to account for the number of "extra" food desired outside of the amount required to make the other food.
   This ends up looking like:
   Dictionary inner in outerVertex values:
   foreach(key in inner)
   ingredientCount[outerVertex]+= ingredientCount[key] * inner[key]
   (out of foreachloop) ingredientCount[outerVertex] += food[outerVertex];
   The final step, now that all ingredients are stored in inGredientCount is to iterate through ingredientCount from i=0 to N and print the value in each index with spaces in between
   This Algorithm should run in approximately O(V+E) time due to only using for loops which are bounded by the number of Vertices and eges, with each vertex only visited a single time

2. (Exact Paths)

Assumptions: Let the length of a given edge between vertices v and w be given by $\ell(v \to w)$ Let the target length be L AND current cumulative Length on a path be M

(a) EP(L,s, M)= $\begin{cases} True & M = L \\ True & EP(L, w, \ell(v \to w) + M) = True \text{ and } v \to w \epsilon E \\ False & otherwise \end{cases}$

The recursive version of this algorithm works by performing a DFS/topological sort, then traversing the graph in that order beginning with S. It looks at every edge connecting S to another vertex, every edge connecting those to other vertices, etc. Marking each as visited along the way. During each of these steps, it adds the length/weight of that edge to a total path Length that is passed into the recurrence. If at any point the Length M equals the desired length L, the algorithm immediately breaks/stops and returns true. Otherwise, if at no point in this process a length of L is found, then the algorithm instead returns false after all paths starting from vertex s have been explored. This algorithm can be memoized with an array of Size 2V OR by storing the answers directly in the Node/vertices of the graph, depending on preference, in which the total length of edges from that vertex to s is stored(these can be stored WHILE performing topological sort/DFS). We then can iterate through this array in reverse topological order and simply compare the total length at each index M to the desired length L, returning true if we find it at any point. This works because seeing that the sub-problem of any given vertex is stored in the array is equivalent to "marking" that node in a typical DFS, and if True exists in any point in the Array. This gives us a run time of O(V+E),which is linear and dependent on the number of edges and Vertices as the graph needs to only be traversed with DFS a single time to fill the memo

(b) EP(L,s, M)= $\begin{cases} 1 & M = L \\ 0 & !v \to w \epsilon E(\text{V has no outgoing edges}) \\ EP(L, w, \ell(v \to w) + M) + EP[i] & \text{if } v \to w \epsilon E \end{cases}$

(it's assumed EP[i] is the current count) The algorithm works similarly to the above(e.g sorting in topological order, traversing in that order starting with S and filling in the memo with information, etc.), however, instead of immediately breaking and returning true when finding a path of Length L, the algorithm instead returns 1, or in other words, increments the total count. If M != L then the algorithm continues to search down that path until it either finds where M=L or otherwise finds that $M > L$ or there is no additional edges to explore, in which case it returns 0. This causes the top level of the recurrence to return the sums down each path by using a foreach loop looking at edges at each vertex. Memoizing this, we do as the above in part aand traverse it in the same order, however, we can merely update a total count int EP[s] every time we find a valid path M=L and at the end, return EP[s] for the final answer, this algorithm should run in O(V+E) time, as the only thing iterated through is each vertex and Edge once, and calculations can be performed concurrently with topological sorting

(c) All you would have to do is change the starting index S to whatever vertex you desire. Since The algo in B already topologically sorts, it would simply explore the paths of the new provided vertex and return similar results as those from part B as/if it finds cases where M=L in any of these paths.just like the other two this will run at O(V+E) time, as the only thing iterated through is each vertex and Edge once, and calculations can be performed concurrently with topological sorting. Of note, you could also adapt this algorithm to find the total number of paths L in the entire graph by starting from the vertex at the top of the topological order and resetting the total length of a given path and adding a count any time you find a point M=L