Jacob Hreshchyshyn
Rithvik Arun
D’Vonye Jackson

MAT 243 Project 6

For this project, we are given a Hasse diagram that describes the process of preparing a Chinese meal. The diagram appears as follows:

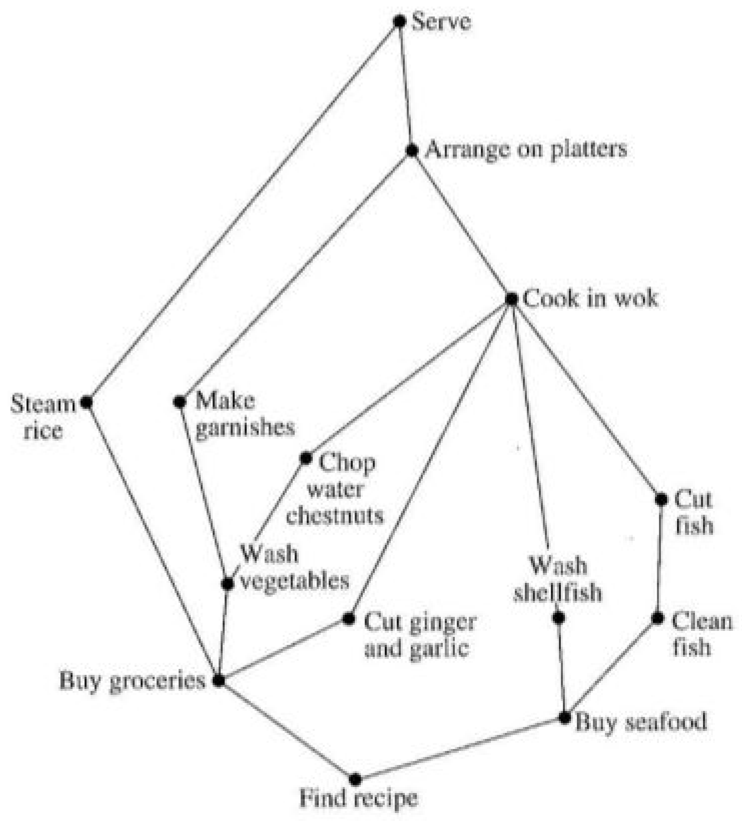

Figure 1 (Rosen 2011)

We are also given a possible ordering in which the process of preparing the meal can be carried out. The possible schedule is as follows:

Find Recipe < Buy Seafood < Buy Groceries < Wash Shellfish < Cut Ginger and Garlic < Clean Fish < Steam Rice < Cut Fish < Wash Vegetables < Chop Water Chestnuts < Make Garnishes < Cook in Wok < Arrange on Platter < Serve.

Given this information, we are tasked with finding three other orderings, determine if one of those orderings is a better way of preparing the meal, and explain why. Finally, we are asked to determine the total number of possible orderings for this diagram.

To begin, the diagram indicates that, before the meal can be served, the rice must be steamed and the other ingredients must be arranged on platters. Before the ingredients can be arranged on platters, they must be cooked in a wok and garnishes must be made. Before garnishes can be made, the vegetables must be washed, and before the other ingredients are cooked in a wok, the water chestnuts must be chopped, the shellfish must be washed, and the fish must be cut. Before garnishes can be made and water chestnuts be chopped, the vegetables must be washed. Before the fish is cut, the fish must be cleaned. Before the rice is steamed, the vegetables are washed, and the ginger and garlic are cut, the groceries need to be purchased. Before the shellfish is washed and the fish is cleaned, the seafood must be purchased. Finally, before the groceries are purchased and the seafood is purchased, the recipe must be found.

While not entirely evident in the above description, one can observe in the diagram that certain tasks are dependent on the completion of other tasks, which may have multiple dependencies themselves. Until each dependency is resolved, it will be impossible to begin work on those tasks.

To help organize these dependencies, we can organize tasks into various levels, which in this instance, start from a single base task and branch out into other dependencies. To do so, we start with the single base taks, Find recipe. We can put this task into Level 1. From Find Recipe, two lines are drawn to two separate tasks, Buy groceries and Buy seafood. We can group these tasks into Level 2. Next, the options that are available from Buy groceries and Buy seafood are Steam rice, Wash vegetables, Cut ginger and garlic, Wash shellfish, and Clean fish. These options can be put into Level 3.

We can see that one of our options, Steam rice, is directly connected to the task Serve. However, Serve will not be put into Level 4 since Serve is dependent on another task, Arrange on platters, which has dependencies of its own. Therefore, we continue searching for connections to tasks that do not have other dependencies. For similar reasons, we will not place Cook in wok in Level 4 despite Cut ginger and garlic's direct connection to that task due to Cook in wok's dependencies.

The next options available are Make garnishes, Chop water chestnuts, and Cut fish. These will be put into Level 4. Now that these Level 4 tasks have been observed, the dependencies for the Cook in wok task have been satisfied. As a result, we can put the Cook in wok task in Level 5. This resolves the dependency for the Arrange on platters task, so we can put this task in Level 6. Finally, with the other dependencies resolved, we place the Serve task in Level 7. Figure 2 demonstrates the described organization of the diagram's tasks and their dependencies.

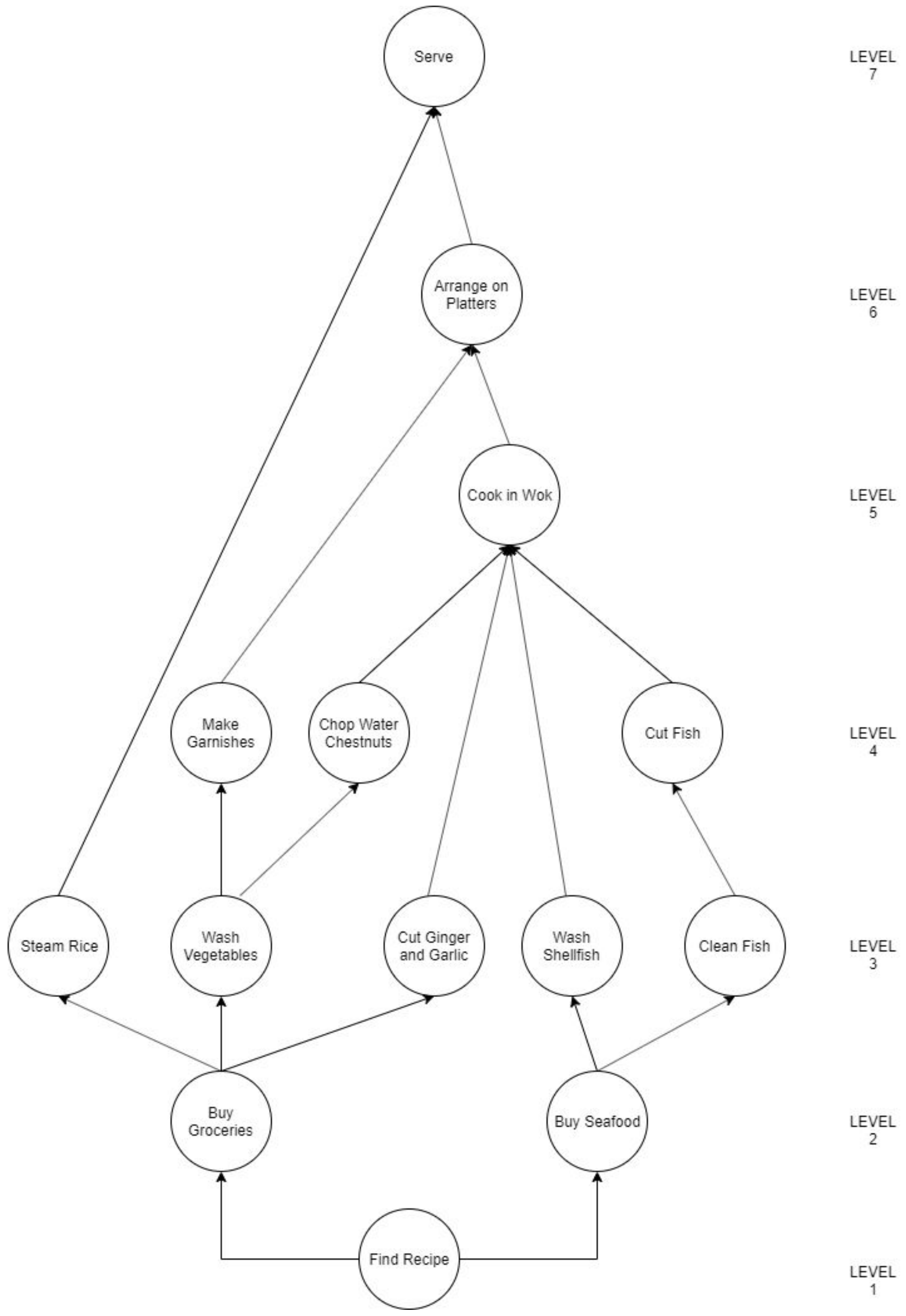

Figure 2

Now that the Hasse diagram has been organized based on the dependencies of each task, one can observe that, given the ordering provided by the project, one can order the process of preparing the meal by jumping between the dependency levels of the diagram in Figure 2. The ordering provided by the project demonstrates this by Cutting the fish before Washing the vegetables, the former task being on Level 4 and the latter being on Level 3. One can then observe that as long as all the dependencies for a task have been resolved, that task can be performed.

With this information, we can now begin to create three more possible orderings for the preparation of the meal.

Ordering 1:

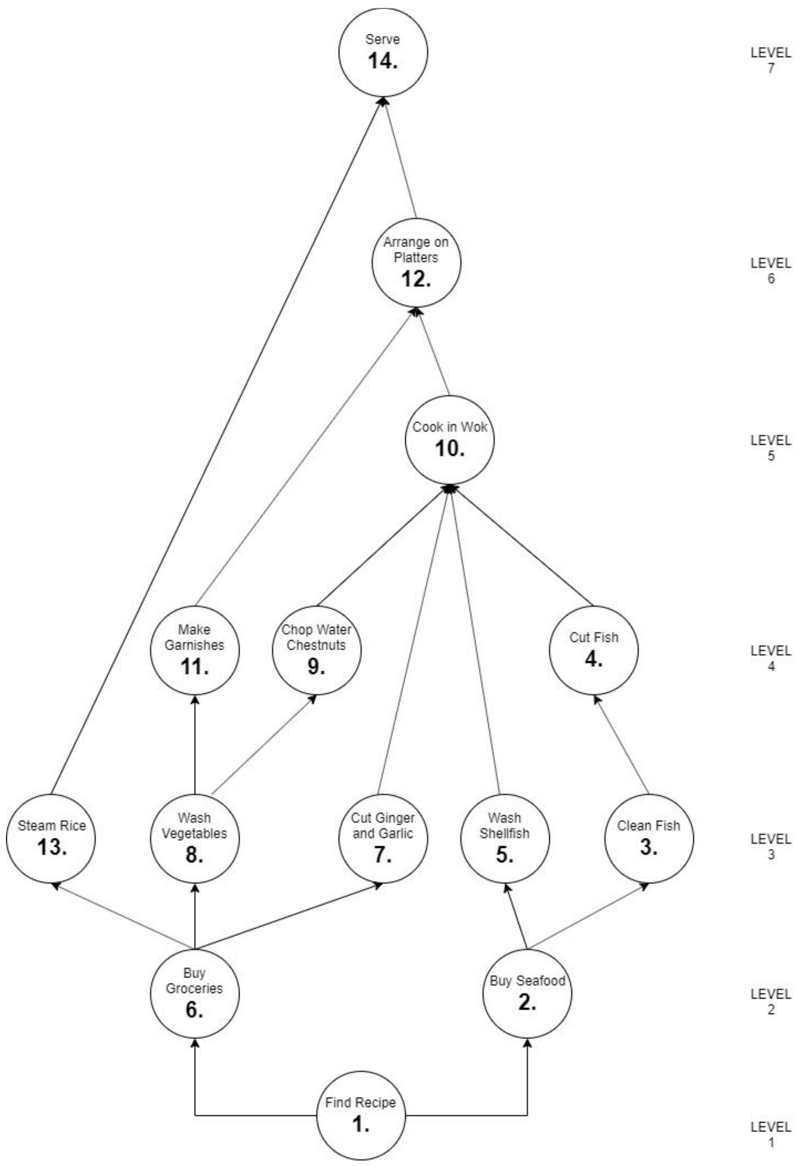

Figure 3

The following describes the ordering of the diagram in Figure 3.

Find Recipe < Buy Seafood < Clean Fish < Cut Fish < Wash Shellfish < Buy Groceries < Cut Ginger and Garlic < Wash Vegetables < Chop Water Chestnuts < Cook in Wok < Make Garnishes < Arrange on Platter < Steam Rice < Serve

Note that this ordering attempts to complete as many tasks on the right side of the diagram as possible. Task completion is only prevented when a task has multiple dependencies, i.e. has more than one previous task connected to it.

Ordering 2:

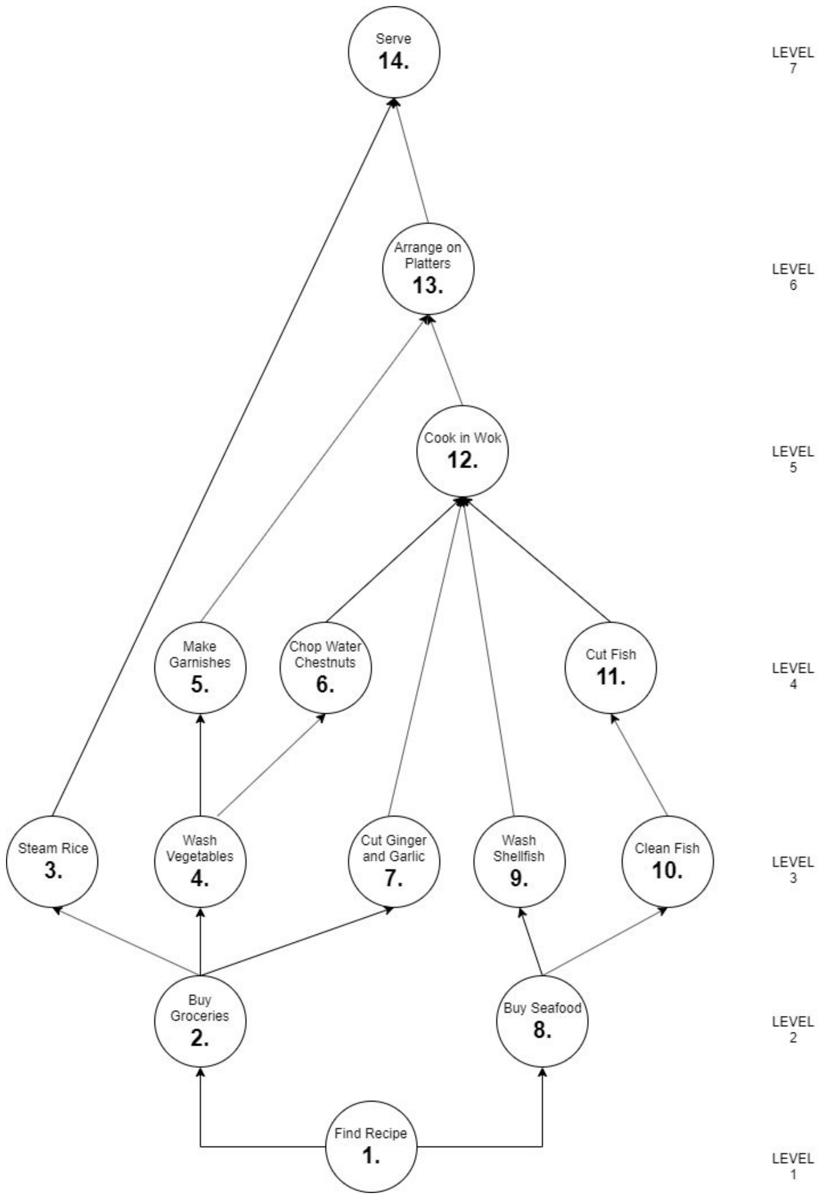

Figure 4

The following describes the ordering of the diagram in Figure 4.

Find Recipe < Buy Groceries < Steam Rice < Wash Vegetables < Make Garnishes < Chop Water Chestnuts < Cut Ginger and Garlic < Buy Seafood < Wash Shellfish < Clean Fish < Cut Fish < Cook in Wok < Arrange on Platter < Serve

This ordering takes the opposite approach of the first ordering by attempting to complete tasks starting from the left side of the diagram. If a task does not have multiple dependencies, that is the next task to be completed. If a task does have multiple dependencies, the ordering begins to look for previous tasks whose dependencies are resolved and have yet to be completed to attempt to resolve the dependencies of later tasks.

Ordering 3:

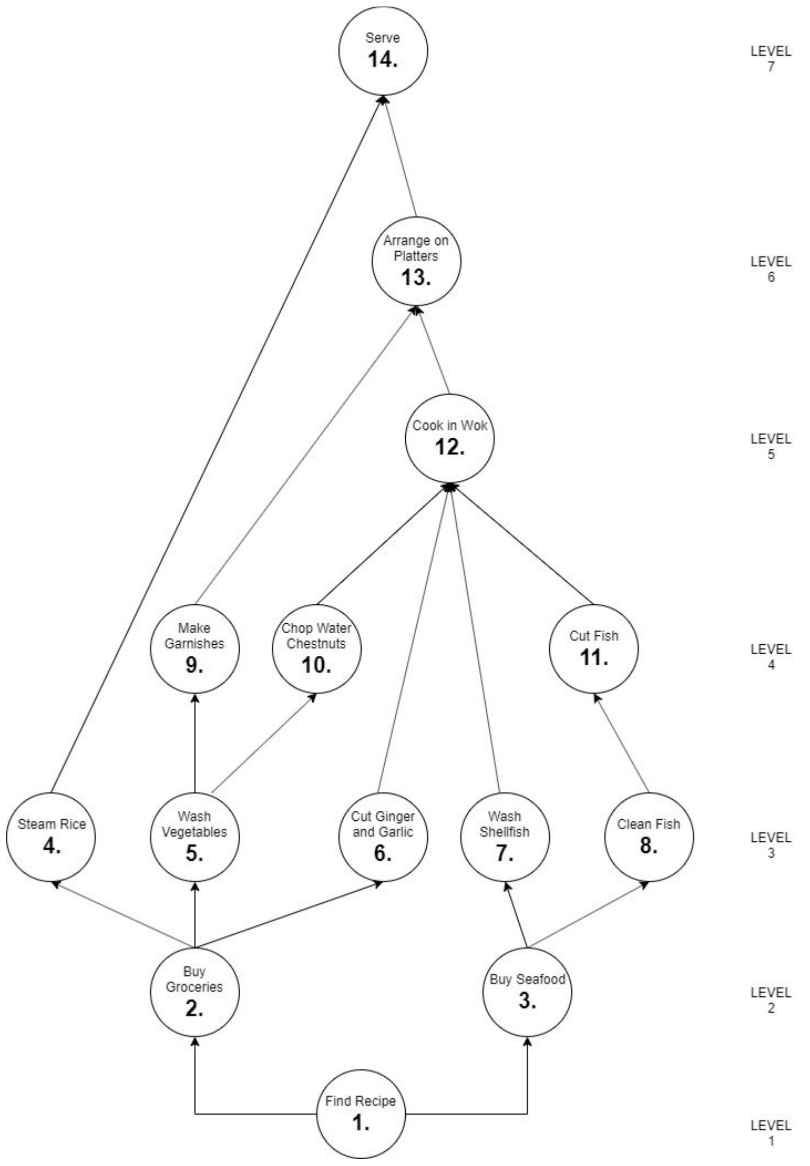

Figure 5

The following describes the ordering of the diagram in Figure 5.

Find Recipe < Buy Groceries < Buy Seafood < Steam Rice < Wash Vegetables < Cut Ginger and Garlic < Wash Shellfish < Clean Fish < Make Garnishes < Chop Water Chestnuts < Cut Fish < Cook in Wok < Arrange on Platter < Serve

This ordering works by completing tasks by levels from left to right. Once every task on a level has been completed, the leftmost task on the next level is completed. In preparing the meal this way, every dependency is accounted for before advancing to the next level. This ordering eliminates the need to go back to a previous level to search for uncompleted dependencies. Because no back-tracking is required using this ordering, we believe that of the four examined orderings in the project, the ordering described in Figure 5 is the best.

We now proceed to determine the total number of possible orderings for the process of producing the Chinese meal. We can attempt to encode a solution which will provide the number of orderings through an exhaustive process. To do so, we can consider the process of finding possible orderings through the use of a Topological Sort. As indicated in Chapter 9.6 of the seventh edition of Rosen's discrete mathematics textbook, it is important to identify the minimal element in a graph, that is, the element in a graph that has no dependencies. These dependencies are also called indegrees, which are represented as lines that point towards graph elements.

If the graph begins with one element with no indegrees, as with the Hasse diagram provided in the project, then that element is selected as the first element of the ordering. Once identified, the element with no indegrees is removed from the graph along with any lines directed away from that element. This alters the graph in such a way that more minimal elements can appear in the graph. When multiple minimal elements exist in a graph, multiple orderings arise. An example of these multiple orderings is given in Figure 6.

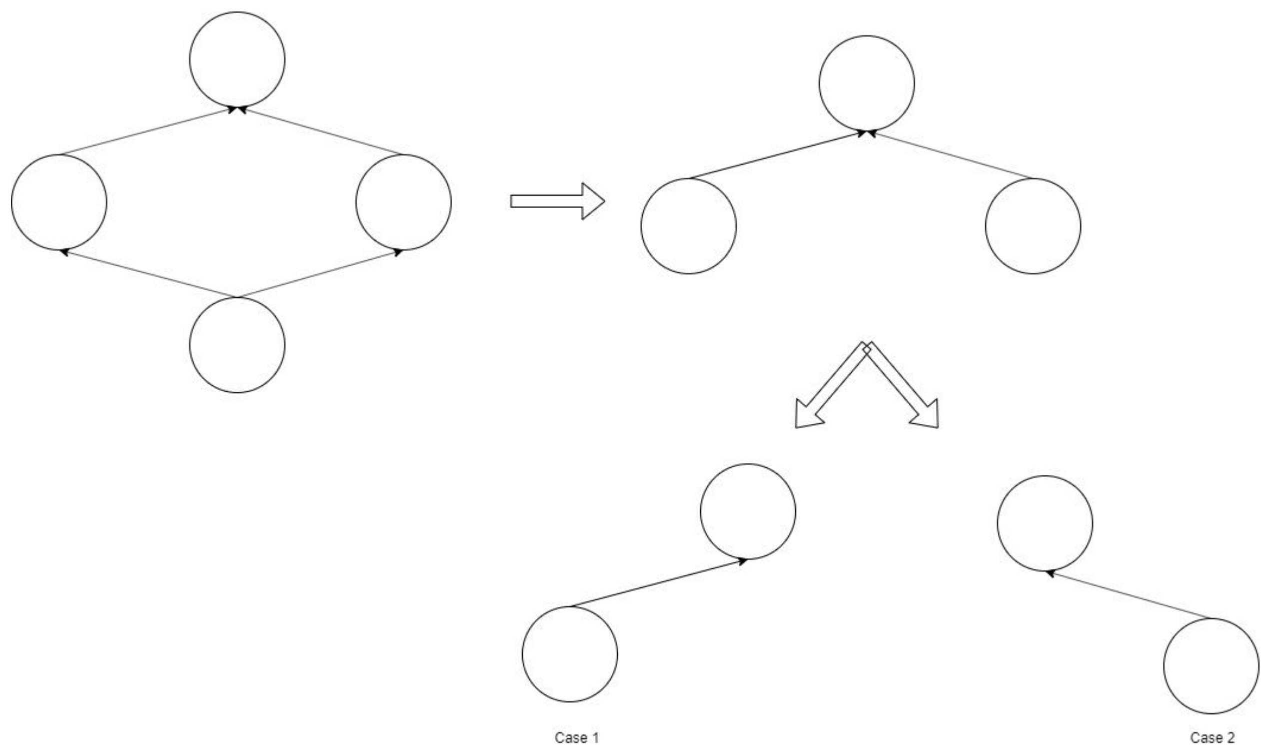

Figure 6

To comb through the given Hasse diagram to find each of these orderings, we used the following Java code in Figure 7, which we modified to give the number of possible orderings.

```
import java.util.*;

class Graph {
   int V; // No. of vertices
   static int count = 0; // tracks number of orderings
   List<Integer> adjListArray[];

   public Graph(int V) {

      this.V = V;

      @SuppressWarnings("unchecked")
      List<Integer> adjListArray[] = new LinkedList[V];

      this.adjListArray = adjListArray;

      for (int i = 0; i < V; i++) {
         adjListArray[i] = new LinkedList<>();
```

```
    }
}
// Utility function to add edge
public void addEdge(int src, int dest) {

    this.adjListArray[src].add(dest);

}

// Main recursive function to print all possible
// topological sorts
private void allTopologicalSortsUtil(boolean[] visited,
                    int[] indegree, ArrayList<Integer> stack) {
    // To indicate whether all topological are found
    // or not
    boolean flag = false;

    for (int i = 0; i < this.V; i++) {
        // If indegree is 0 and not yet visited then
        // only choose that vertex
        if (!visited[i] && indegree[i] == 0) {

            // including in result
            visited[i] = true;
            stack.add(i);
            for (int adjacent : this.adjListArray[i]) {
                indegree[adjacent]--;
            }
            allTopologicalSortsUtil(visited, indegree, stack);

            // resetting visited, res and indegree for
            // backtracking
            visited[i] = false;
            stack.remove(stack.size() - 1);
            for (int adjacent : this.adjListArray[i]) {
                indegree[adjacent]++;
            }

            flag = true;
        }
    }
    // We reach here if all vertices are visited.
    // So we print the solution here
    if (!flag) {
        stack.forEach(i -> System.out.print(i + " "));
        System.out.println();
        count++;
    }

}

// The function does all Topological Sort.
// It uses recursive alltopologicalSortUtil()
public void allTopologicalSorts() {
    // Mark all the vertices as not visited
    boolean[] visited = new boolean[this.V];

    int[] indegree = new int[this.V];

    for (int i = 0; i < this.V; i++) {

        for (int var : this.adjListArray[i]) {
            indegree[var]++;
        }
    }

    ArrayList<Integer> stack = new ArrayList<>();

    allTopologicalSortsUtil(visited, indegree, stack);
}

// Driver code
public static void main(String[] args) {



    Graph graph = new Graph(14);
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 3);
    graph.addEdge(1, 4);
    graph.addEdge(1, 5);
    graph.addEdge(2, 6);
    graph.addEdge(2, 7);
    graph.addEdge(3, 13);
    graph.addEdge(4, 8);
    graph.addEdge(4, 9);
```

```
        graph.addEdge(5, 11);
        graph.addEdge(6, 11);
        graph.addEdge(7, 10);
        graph.addEdge(8, 12);
        graph.addEdge(9, 11);
        graph.addEdge(10, 11);
        graph.addEdge(11, 12);
        graph.addEdge(12, 13);

        System.out.println("All Topological sorts");
        graph.allTopologicalSorts();
        System.out.println(count);
    }
}
```

Figure 7

It is worth noting that this code assigns a number to each of the tasks from 0 to 13 so that the generation of orderings can be executed. The number assignment is as follows:

Find recipe - 0
Buy groceries - 1
Buy seafood - 2
Steam rice - 3
Wash vegetables - 4
Cut ginger and garlic - 5
Wash shellfish - 6
Clean fish - 7
Make garnishes - 8
Chop water chestnuts - 9
Cut fish - 10
Cook in wok - 11
Arrange on platters - 12
Serve - 13

The code given in Figure 7 works first by establishing the blueprints for the graph itself. There will be a number of vertices, or elements, in the graph as well as a global count variable that tracks the number of possible orderings in the graph. There is also an array of integer lists called adjListArray that will help keep track of elements adjacent to an element visited in the program. This array will be set to a linked list array that has as many elements as the elements in the graph. Then, each element of adjListArray is made to contain an individual linked list.

Next, the program includes a method that allows elements in the graph to be mapped to other elements. This method, addEdge, does so by going to the specified linked list in adjListArray and appending an integer to that list (note: this program requires that the elements in the graph be labelled from 0 to n - 1. In this case, the first element is 0 and the last element is 13).

Next, the program utilizes two methods to produce all the possible orderings. Starting with allTopologicalSorts, the program creates an array of booleans to help keep track of elements that have already been visited during traversal of the graph. The array is the size of the number of elements in the graph, which would be 14 in this case. There is also an integer array called indegree, which will help keep track of the number of indegrees each element has. The size of the array will be the same as the number of elements in the graph so that each element of the array stores the number of indegrees for each element.

Having declared the indegree array, a for loop is used to examine each element of the graph. Because each element has a linked list containing other elements mapped to itself, a for-each loop is used to comb through each linked list and search for other integers in each linked list, indicating that an element with a linked list containing numbers has indegrees associated with it. The number of indegrees each element has is stored in the indegree array at the index that matches the index of the element in the adjListArray.

Now that the elements of a graph as well as the indegrees of those elements can be stored, the program creates an integer array list called stack, which will contain a possible ordering that corresponds with the graph. Finally, the allTopologicalSorts method calls the recursive function allTopologicalSortsUtil, which has the array of booleans, the array of indegrees, and the stack array list as parameters.

allTopologicalSortsUtil begins by setting a boolean called flag to false, which will be set to true once all the orderings have been found. Next, a for loop that iterates as many times as there are elements begins. Using the passed in visited boolean array and indegree integer array, it checks whether each element, starting from the first, was not yet checked and if the element has no indegree. If both conditions are met, then the program will mark the boolean at the index that matches the element as true. Within that same block, the program will also add the index of the element into the passed in stack integer array list. A for-each loop then checks the element whose index matches that of i and reduces the indegree number for the adjacent elements. With the updated boolean array, indegree array and stack, these new parameters are passed into the recursive function call.

Once every element has been visited through these recursive calls, with each call adding minimal elements to the stack, the contents of the stack, which demonstrates the first ordering, is output and the count for the total number of orderings is incremented by one. This marks the end of the stack frame (not to be confused with the stack array list), meaning that the program continues past the recursive function call and resets the values of visited, indegree, and the stack array list so that the program can continue to find other possibilities through recursion. This eventually leads to every flag boolean to be set to true, which means that every possible ordering has been exhausted.

On executing the program, we received 37,674 as the total number of possible orderings for preparing this Chinese meal. This large number makes sense when considering the many different ways to remove tasks such that multiple tasks can exist without any dependencies or indegrees. This inevitably leads to many different orders in which one can remove dependencies and tackle tasks without dependencies.

References:

Reference code for finding all possible orderings of the given Hasse Diagram. I modified it by remapping the elements and adding a global count variable that tracks the total number of orderings. Also source for term “indegree”.
https://www.geeksforgeeks.org/all-topological-sorts-of-a-directed-acyclic-graph/

Reference video for better visualizing how to find orderings using topological sorting. Also source for term “indegree”.
https://www.youtube.com/watch?v=a3UDL-v6-44