```
main(G):
    G_r = graph_reverse(G) //flip all edges in graph, O(m)

    G_sinks = DFS_all(G_r).sources //black box DFS_all to find sources from G_r

    return maximum of (find_longest_path for each of G_sinks)

graph_reverse(G):  //O(m)
    for all edges in G:
        edge(u,v) = edge(v,u)
    return G

find_longest_path(G, v):
    if v has no children:
        return v.time

    return v.time + max(find_longest_path for each of v.children)
```

```
1   main(G):
2       G_r = graph_reverse(G) //flip all edges in graph, O(m)
3
4       return find_longest_path(G_r, v) - v.time for each vertex v in G
5
6   graph_reverse(G):  //O(m)
7       for all edges in G:
8           edge(u,v) = edge(v,u)
9       return G
10
11  find_longest_path(G, v):
12      if v has no children:
13          return v.time
14
15      return v.time + max(find_longest_path for each of v.children)
```

```
main(G, overall_completion_time):
    for v in G {
        G_vsubgraph = subgraph of G including only v and all its descendents //O(n)

        time_needed = problem_A(G_vsubgraph) //main function from problem A

        latest_times[v] = overall_completion_time - time_needed
    }
    return latest_times
```

Worked with Jeremy Varghese.

24 (100 PTS.) Run DAG run!

Suppose we are given a DAG G with n vertices and m edges. The vertices represent jobs and whose edges represent precedence constraints. That is, each edge (u, v) indicates that job u must be completed before job v begins. Each node v also has a weight $T(v)$ indicating the time required to execute job v.

For all the following questions, you can assume that you have as many processors as you want (that work independently in parallel).

24.A. (30 PTS.) Describe an algorithm to determine the shortest interval of time in which all jobs in G can be executed.

24.B. (40 PTS.) Suppose the first job starts at time 0. Describe an algorithm to determine, for each vertex v, the earliest time when job v can begin.

24.C. (30 PTS.) Now describe an algorithm to determine, for each vertex v, the latest time when job v can begin without violating the precedence constraints or increasing the overall completion time (computed in part (A)), assuming that every job that has precedence on v starts at its earliest start time (i.e., computed in part (B)).

24

Solution:

1. **Idea:** We start by reversing direction of all edges in given graph G.

   We then run DFS on all vertices to find sources within our reversed graph by marking each vertex with time done and visited markers. We pick max time done valued nodes in each DFS subtree add it to source list size k.

   For each source in our list, we want to find the max value of a traversal to a sink. We do this by recursively calling each of the current nodes children, starting from our source node and adding the time value from the current node. The path we take will be the child with the max time value. Once we reach a node with no children we have our final minimum time value of an execution order.

   After each source from our list goes through this, we have minimum execution times for all of them. Because we reversed the graph at first, this represents the time needed to execute all the way to each sink. We then return the maximum execution time of all sources.

   Psuedocode: See image 1.

   Analysis: Our algorithm reverses the given graph G and then calls DFSAll to find sources. Reversing takes O(m) time as we simply iterate through the list of graph objects and reverse each outgoing edge. We use the unmodified version of DFSAll that is discussed in lecture which we found to take O(m+n). This bounds our runtime as O(m+n).

2. **Idea:** The approach we took for this part was similar to part A. We still reverse the graph at first. Instead of finding the sinks, we simply call find longest path from each vertex graph g which gives us the minimum time needed until the current nodes task can finish. We modify our algorithm from part 1 for the current problem by not adding the current node's value therefore giving us the time the job can begin. Then by assigning the value of the function call to the current node, we get the earliest time when its job can begin.

   Psuedocode: See image 2.

   Analysis: This approach is very similar to the first part with a single line difference. It still uses a graph reversal and DFSAll, giving the same run time of O(m+n) for the same reasons given above.

3. **Idea:** Utilizing our parts A and B, we first call our algorithm in A. This gives us the minimum time required to execute all tasks. Then for each node u, we find a subgraph which contains all nodes that can be reached from the current node. Within the subgraph of v, we find the longest execution path needed to reach a sink by calling the algorithm in A on the entire subgraph. We then subtract the value of the subgraph's problem A call from the overall graph's problem A call. This gives us the latest start time the current node can begin without increasing overall execution time.

   Psuedocode: See image 3.

   Analysis: Our for loop goes through m iterations of subproblem A and a subgraph traversal. The subgraph traversal would take at most O(n) steps to Earlier we found that subproblem A would take us O(n+m). This gives us a total run time of $O(n * (n + m))$.
