Algorithm Complexity Analysis

• US17:

This initial user story aims to create the graph with the activities.
For each activity obtained from activityRepository.getAllActivities(), a new ActivityInfo is created and mapped.
Considering the number of activities N, the complexity of this part is O(N).
Then it is necessary to traverse each entry in the predecessorsActivityMap map (which contains a mapping of activities to lists of predecessors), and for each one a relationship is created between activities (edges), and considering the number of predecessors per activity P, then the complexity when traversing the map is O(N*P).
The same O(N*P) complexity occurs to discover which activities should be linked to the end point.
Thus, the total complexity to create the graph is O(N*P).

• US18:

The objective is to detect a cycle in the graph and used a depth-first search (DFS) algorithm. Thus, the complexity depends on the number of vertices of the graph (V) and the complexity of the dfs – O(V)*O(dfs).
The dfs in the worst case goes through all vertices and all edges – O(V+E), because each vertex/edge is visited at most once.
Thus, the algorithm to detect cycles has complexity O(V*(V+E)).

• US19:

This user story performs a topological order in a directed graph, and for this purpose, depth-first search (DFS) algorithm was used for each unvisited vertex. When traversing each vertex (for (V vertex : graph.vertices())) the complexity is O(V) but checking whether the vertex was visited (if (!visited.contains(vertex))) takes place in O(1) due to the use of HashSet. If it has not been visited, the method that contains the dfs for that vertex is called.
In dfs, as each vertex and edge is visited at most once, the complexity is O(V+E).
In other words, as before the dfs call we have a check to see if the vertex has already been visited, the total complexity ends up being dominated by the dfs, thus being O(V+E).

• US20:

To calculate the times, the topological order is initially called (which, as already mentioned, has a complexity of O(V+E).
Thus, this is traversed (O(V)), and for each vertex the incomingVertices method is used, which returns the vertices that have edges directed to the current vertex which, in the worst case, is proportional to the number of edges (E), being the complexity then O(V+E).
This will occur twice, first to calculate ES and EF, and then to calculate LS and LF, not differing in complexity since one follows the order of the topological sort and the other just follows the reverse order, which means that in the it passes through all vertices V.
Finally, the slack of each vertex is calculated, which simply involves iterating over all vertices of the graph and performing a constant operation – O(V).
Thus, the total complexity is: O(V+E)+O(V+E)+O(V+E)+O(V)=O(V+E)

• US21:

Since the objective is to generate a file with data on activities and their precedents, it iterates over all vertices of the graph and their respective edges, so its complexity is O(V+E).

• US22:

This user story has two distinct objectives – calculating total time and discovering critical paths.

Regarding the total time, it goes through all the vertices of the graph and updates the time if it is greater, that is, the complexity is O(V).

Regarding critical paths, initially it is necessary to find the final vertex and to do this it is necessary to traverse all (O(V)), then all the vertices linked to this final vertex are traversed and the findPath method is called, which in the worst case is O (V) since each vertex is traversed once. Thus, the complexity is $O(V^2)$.

• US23:

The objective is to calculate the dependency level of each vertex based on the number of predecessors and sort based on that.
Thus, all vertices of the graph are traversed (O(V)) and all predecessors of each vertex are collected (O(E)).
It is subsequently inserted into a heap priority queue, and this has an insertion complexity of O(logV).
In conclusion, the complexity of this user story is O(E + V*logV).

• US24:

Initially we have a delayedActivities map that contains the activities with their respective delays. Iterating over the map keys has O(A) complexity, where A is the number of activities in the delay map. For each activity with delays, the function traverses all vertices of the graph (O(V)) in search of the corresponding vertex. Therefore, to update the graph with the delay times, O(A×V) is necessary.

Later we have calls to other user stories:

• US22 – calculateTotalTimeDuration – calculate total time before delays – O(V)

• US20 – calculateTimes – recalculate the start and end times of activities - O(V+E)

• US22 – calculateTotalTimeDuration – recalculate total time after delays – O(V)

• US22 – generateCriticalPathsList – recalculate the critical paths of the graph - O(V2)

In addition to the complexities of each of the called features, a subtraction of the total time after minus the total time before delays is made – O(1).