COMPLEXITY

USEI17

```
/**
 * Builds the PERT/CPM graph.
 */
public void buildPERT_CPM() {
    // Add the "START" node
    activitiesPERT_CPM.clear();
    ActivitiesMapRepository activitiesMapRepository =
Instances.getInstance().getActivitiesMapRepository();
    activities = activitiesMapRepository.getActivitiesMapRepository();
    pert_CPM.addVertex("START");
    activitiesPERT_CPM.put("START", new Activity("START", "START", 0, 0, new
ArrayList<>()));

    for (Activity activity : activities.values()) {
        // Create the node label with duration
        String nodeLabel = activity.getActId() + " (" + activity.getDurationWithUnit() +
")";
        pert_CPM.addVertex(nodeLabel);

        // Connect "START" to activities without predecessors
        if (activity.getPrevActIds().isEmpty()) {
            pert_CPM.addEdge("START", nodeLabel, "0");
        }

        // Connect activities to their predecessors
        for (String dependencyId : activity.getPrevActIds()) {
            Activity dependencyActivity = activities.get(dependencyId);
            if (dependencyActivity != null) {
                String dependencyLabel = dependencyId + " (" +
dependencyActivity.getDurationWithUnit() + ")";
                pert_CPM.addEdge(dependencyLabel, nodeLabel, "0");
            }
        }

        activitiesPERT_CPM.put(activity.getActId(), activity);

        if (activity.getPrevActIds().isEmpty()) {
            List<String> modifiablePrevActIds = new ArrayList<>(activity.getPrevActIds());
            modifiablePrevActIds.add("START");
            activity.setPrevActIds(modifiablePrevActIds);
        }
    }

    // Add the "END" node
    pert_CPM.addVertex("END");
    activitiesPERT_CPM.put("END", new Activity("END", "END", 0, 0, new ArrayList<>()));

    // Connect all activities without successors to the "END" node
    for (String vertex : pert_CPM.vertices()) {
        String actId = vertex.split(" ")[0];
        boolean hasSuccessors = false;
        // A node without successors has an out-degree of 0
        if (pert_CPM.outDegree(vertex) == 0 && !vertex.equals("END") &&
!vertex.equals("START")) {
            // Connect to "END"
            pert_CPM.addEdge(vertex, "END", "0");
            if (activitiesPERT_CPM.containsKey(actId)) {

activitiesPERT_CPM.get("END").getPrevActIds().add(activitiesPERT_CPM.get(actId).getActId())
```

```
;
            }
        }
        for (String actId2 : activities.keySet()) {
            Activity activity = activities.get(actId2);
            if (activity.getPrevActIds().contains(actId)) {
                hasSuccessors = true;
            }
        }
        if (!hasSuccessors && !vertex.equals("END") && !vertex.equals("START") &&
activitiesPERT_CPM.containsKey(actId) &&
!activitiesPERT_CPM.get("END").getPrevActIds().contains(actId)) {
            // Connect to "END"

activitiesPERT_CPM.get("END").getPrevActIds().add(activitiesPERT_CPM.get(actId).getActId())
;
        }
    }
}
```

Time Complexity `buildPERT_CPM()`: O(V^2 + V * P)

- The outer loop iterates over all activities, leading to a complexity of **O(V)**, where V is the number of activities.
- For each activity, the predecessors are processed, leading to **O(P)** operations per activity, where P is the average number of predecessors.
- The method then iterates over all vertices to connect to the "END" node, contributing an additional **O(V)** complexity.
- Finally, it iterates over all activities again to check for successors, which adds another O(V).
- Combining all, the overall time complexity is **O(V^2 + V * P)**.

USEI18

```
/**
 * Checks if there are circular dependencies in the graph.
 *
 * @return true if a cycle is detected, false otherwise.
 */
public boolean hasCircularDependencies() {
    Set<String> visited = new HashSet<>();
    Set<String> recursionStack = new HashSet<>();

    for (String vertex : pert_CPM.vertices()) {
        if (detectCycle(vertex, visited, recursionStack)) {
            return true; // Cycle detected
        }
    }
    return false; // No cycle detected
}
```

Time Complexity `hasCircularDependencies()`: O(V * (V + E))

- The method iterates over all vertices in the graph, contributing **O(V)** complexity.
- For each vertex, `detectCycle()` is called, which has a time complexity of **O(V + E),** where E is the number of edges.
- Combining both, the worst-case time complexity is **O(V * (V + E)).**

```
private boolean detectCycle(String vertex, Set<String> visited, Set<String> recursionStack)
{
    if (recursionStack.contains(vertex)) {
        return true; // Cycle detected
    }

    if (visited.contains(vertex)) {
        return false; // Already processed, no cycle
    }

    visited.add(vertex);
    recursionStack.add(vertex);

    // Check all adjacent vertices
    for (String neighbor : pert_CPM.adjVertices(vertex)) {
        if (detectCycle(neighbor, visited, recursionStack)) {
            return true; // Cycle detected in subgraph
        }
     }
     recursionStack.remove(vertex);
    return false; // No cycle detected
}
```

Time Complexity `detectCycle()`: O(V + E)

- The method performs a recursive DFS, visiting each vertex once, contributing **O(V).**
- For each vertex, it checks all adjacent vertices, which results in **O(E)** complexity.
- Combining both, the time complexity is **O(V + E).**

```java
/**
 * Validates the graph.
 * @return true if the graph is valid, false otherwise.
 */
public boolean validateGraph() {
    boolean hasCiruclar=false;
    if (hasCircularDependencies()) {
        hasCiruclar=true;
    }
    else {
        hasCiruclar=false;
    }
return hasCiruclar;
}
```

Time Complexity `validateGraph()`: O(V * (V + E))

- The method calls `hasCircularDependencies()`, which has a time complexity of **O(V * (V + E)).**
- Therefore, the overall time complexity is **O(V * (V + E)).**

USEI19

```java
public List<String> topologicalSort() {
       if (hasCircularDependencies()) {
        throw new IllegalStateException("The graph has circular dependencies.");
    }
    Stack<String> stack = new Stack<>(); // Stack to store the sorted order
    Set<String> visited = new HashSet<>(); // Set to track visited nodes
    // Iterate over all vertices in the graph
    for (String vertex : pert_CPM.vertices()) {
        if (!visited.contains(vertex)) {
            topologicalSortUtil(vertex, visited, stack);
        }
    }
    // Convert the stack into the result list
    List<String> sortedOrder = new ArrayList<>();
    while (!stack.isEmpty()) {
        sortedOrder.add(stack.pop());
    }
    return sortedOrder;
}
```

Time Complexity `topologicalSort()`: O(V * (V + E))

- The method first checks for circular dependencies, which takes **O(V * (V + E))** time.

- It then iterates over all vertices, contributing **O(V).**
- For each unvisited vertex, `topologicalSortUtil()` is called, which also adds **O(V).**
- Finally, converting the stack to a list adds **O(V).**
- Combining all, the overall time complexity is **O(V * (V + E)).**

```
private void topologicalSortUtil(String vertex, Set<String> visited, Stack<String> stack) {
    // Mark the current vertex as visited
    visited.add(vertex);

    // Recur for all adjacent vertices
    for (String neighbor : pert_CPM.adjVertices(vertex)) {
        if (!visited.contains(neighbor)) {
            topologicalSortUtil(neighbor, visited, stack);
        }
    }

    // Push the current vertex to the stack
    stack.push(vertex);
}
```

Time Complexity `topologicalSortUtil()`: O(V + E)

- The method performs a recursive DFS, visiting each vertex once, contributing **O(V).**
- For each vertex, it explores all adjacent vertices, resulting in **O(E).**
- Therefore, the time complexity is **O(V + E).**

USEI20

calculateTimes()

- **Method Calls:**
  - clearTimes(activities): **O(n)**
  - calculateESEF(activities): **O(n^3)**
  - calculateLSLF(activities): **O(n^3)**
  - calculateSlack(activities): **O(n)**
- **Complexity:** The complexity is determined by the sum of the complexities of these individual methods.
- Combining all, the overall time complexity is **O(n^3)**

```
public void calculateTimes() {
    pert_cpm = Instances.getInstance().getPERT_CPM();
    LinkedHashMap<String, Activity> activities =
pert_cpm.getActivitiesPERT_CPM();
    clearTimes(activities);
    calculateESEF(activities);
    calculateLSLF(activities);
    calculateSlack(activities);
}
```

clearTimes(LinkedHashMap<String, Activity> activities)

- **Operation:** Iterates over all the activities in the LinkedHashMap and clears the values to their default 0.
- **Complexity: O(n)**, where n is the number of activities.

```
private void clearTimes(LinkedHashMap<String, Activity> activities) {
    for (Activity activity : activities.values()) {
        activity.clearTimes();
    }
}
```

calculateESEF(LinkedHashMap<String, Activity> activities)

- **Operation:** Get the start activity so you can calculate from the start as the ES and EF calculations start from the beginning for the calculation we will use a recursive method.
- **Calls calculateESEF(LinkedHashMap<String, Activity> activities)** that has an complexity of **O(n^3)**.
- **Complexity: O(n^3)**

```
private static void calculateESEF(LinkedHashMap<String, Activity>
activities) {
    Activity start = activities.get("START");
    start.calculateESEF(activities);
}
```

calculateESEF(LinkedHashMap<String, Activity> activities)

Logic Breakdown:

1. Handles a special case for the "START" node (**O(1)**).
2. Iterates over the predecessor activities (prevActIds), performing:
    - A lookup in the LinkedHashMap (**O(1)**).
    - A comparison and update operation (**O(1)**).

3. Computes earliestStart as earliestStart equal to the biggest earliest finish (**O(1)**).
4. Computes earliestFinish as earliestStart + duration (**O(1)**).

Complexity:

- Iterates over **n** predecessors: **O(n)**.
- Calls the function calculateESEFRec(LinkedHashMap<String, Activity> activities) **n** times that has a complexity of **O(n^2)** so:
- Total: **O(n^3)**

```
public void calculateESEF(LinkedHashMap<String, Activity> activities) {
    if (actId.equalsIgnoreCase("START")) {
        earliestStart = 0.0;
        earliestFinish = 0.0;
        List<Activity> nextActIds = new ArrayList<>();
        for (Activity activity : activities.values()) {
            if (activity.getPrevActIds().contains(actId)) {
                nextActIds.add(activity);
            }
        }
        for (Activity activity : nextActIds) {
            activity.calculateESEFRec(activities);
        }
    }

}
```

calculateESEFRec(LinkedHashMap<String, Activity> activities)

- **Logic**:
    - **For Loop**: Iterates over prevActIds (predecessor activities), performing constant-time operations. **O(p)** where p ≤ n.
    - **Find Successors**: Loops through all activities to identify successors. **O(n)**.
    - **Recursive Call**: Propagates to successors.
- **Complexity**: **O(n^2)** in the worst case due to nested loops and recursion across the activity graph.

```
private void calculateESEFRec(LinkedHashMap<String, Activity> activities) {
    if (actId.equalsIgnoreCase("END")) {
        for (String prevActId : prevActIds) {
            double prevActFinish =
activities.get(prevActId).getEarliestFinish();
            if (prevActFinish > earliestStart) {
                earliestStart = prevActFinish;
            }
        }
        earliestFinish = earliestStart + duration;
```

```
        return;
    }
    for (String prevActId : prevActIds) {
        double prevActFinish =
activities.get(prevActId).getEarliestFinish();
        if (prevActFinish > earliestStart) {
            earliestStart = prevActFinish;
        }
    }
    earliestFinish = earliestStart + duration;
    List<Activity> nextActIds = new ArrayList<>();
    for (Activity activity : activities.values()) {
        if (activity.getPrevActIds().contains(actId)) {
            nextActIds.add(activity);
        }
    }
    for (Activity activity : nextActIds) {
        activity.calculateESEFRec(activities);
    }
}
```

calculateLSLF(LinkedHashMap<String, Activity> activities)

- **Steps:**
    1. Creates a reversed LinkedList of the activities (**O(n)**).
    2. Gets the activity end form the reversed list (**O(1)**).
    3. Iterates over the activities, calling calculateLSLF on the end activity.
        - If calculateLSLF involves looking up predecessor/successor activities in the LinkedHashMap, the cost is per lookup due to LinkedHashMap's constant-time access property.

- **Calls calculateLSLF(LinkedHashMap<String, Activity> activities)** that has an complexity of **O(n^3).**
- **Complexity: O(n^3)**

```
private static void calculateLSLF(LinkedHashMap<String, Activity>
activities) {
    LinkedList<Activity> reversedActivities = new
LinkedList<>(activities.values());
    Collections.reverse(reversedActivities);
    Activity end = reversedActivities.getFirst();
    end.calculateLSLF(activities);
}
```

calculateLSLF(LinkedHashMap<String, Activity> activities)

- **Operation**: Calls calculateLSLFRec on the **END** activity. This method recursively propagates through all predecessors.
- **Complexity**: **O(n^3)** in the worst case because:
    - At each step, it loops over predecessors (O(p)),
    - Recursively calls calculateLSLFRec for each predecessor.
    - Total cost per activity: **O(n)** × **O(n)** = **O(n^2).**
    - Since recursion may visit every activity: **O(n^3).**

```
public void calculateLSLF(LinkedHashMap<String, Activity> activities) {
    if (actId.equalsIgnoreCase("END")) {
        latestFinish = earliestFinish;
        latestStart = earliestStart;
    }
    List<Activity> prevActIds = new ArrayList<>();
    for (String prevActId : this.prevActIds) {
        prevActIds.add(activities.get(prevActId));
    }
    for (Activity activity : prevActIds) {
        double activityStart = latestStart;
        if (activity.getLatestFinish() > activityStart && activityStart != 
0) {
            activity.setLatestFinish(activityStart);
            activity.setLatestStart(activity.getLatestFinish() - 
activity.getDuration());
        } else if (activity.getLatestFinish() == 0) {
            activity.setLatestFinish(activityStart);
            activity.setLatestStart(activity.getLatestFinish() - 
activity.getDuration());
        }
        activity.calculateLSLFRec(activities);
    }
}
```

calculateLSLFRec(LinkedHashMap<String, Activity> activities)

- **Logic**:
    - **For Loop**: Iterates over prevActIds (predecessor activities), performing constant-time operations. **O(p)** where p ≤ n.
    - **Recursive Call**: Propagates backward through all predecessors.
- **Complexity**: **O(n^3)** in the worst case due to recursive calls and nested iterations over the graph structure.

```
private void calculateLSLFRec(LinkedHashMap<String, Activity> activities) {
    if (prevActIds.isEmpty()) {
        latestFinish = 0;
        latestStart = 0;
```

```
            return;
        }
        List<Activity> prevActIds = new ArrayList<>();
        for (String prevActId : this.prevActIds) {
            prevActIds.add(activities.get(prevActId));
        }
        for (Activity activity : prevActIds) {
            double activityStart = latestStart;
            if (activity.getLatestFinish() > activityStart && activityStart !=
0) {
                activity.setLatestFinish(activityStart);
                activity.setLatestStart(activity.getLatestFinish() -
activity.getDuration());
            } else if (activity.getLatestFinish() == 0) {
                activity.setLatestFinish(activityStart);
                activity.setLatestStart(activity.getLatestFinish() -
activity.getDuration());
            }
            activity.calculateLSLFRec(activities);
        }
}
```

calculateSlack(LinkedHashMap<String, Activity> activities)

- **Operation:** Iterates over all the activities in the LinkedHashMap, calling calculateSlack on each activity.
- **Complexity:** **O(n)**

```
private static void calculateSlack(LinkedHashMap<String, Activity>
activities) {
    for (Activity activity : activities.values()) {
        activity.calculateSlack();
    }
}
```

- **Calls:**

```
public void calculateSlack() {
    slack = latestStart - earliestStart;
}
```

USEI21

```
/**
 * Exports the schedule to a CSV file.
 * @param file File to export the schedule to.
 */
public void exportScheduleToCSV(File file) {

    CalculateTimes calculateTimes = new CalculateTimes();
    calculateTimes.calculateTimes();
    MapGraph<String, String> pert_CPM_without_duration = getPert_CPMWithoutDuration();
```

```
    try (FileWriter writer = new FileWriter(file)) {
        // CSV header
        writer.write("act_id;cost;duration;es;ls;ef;lf;prev_act_id1;...;prev_act_idN\n");

        // Write the data for each activity
        for (String actId : activities.keySet()) {
            Activity activity = activities.get(actId);

            // Get the dependencies of the activity
            Collection<Edge<String, String>> dependencies =
pert_CPM_without_duration.incomingEdges(actId);

            // Convert the dependencies to a string
            String dependenciesStr = (dependencies == null || dependencies.isEmpty())
                    ? "-"
                    : dependencies.stream()
                    .map(Edge::getVOrig) // Get the IDs of the dependencies
                    .reduce((a, b) -> a + ", " + b) // Concatenate the IDs
                    .orElse("-");


            // Write the activity data to the file
            writer.write(String.format("%s;%d;%d;%.1f;%.1f;%.1f;%.1f;%s\n",
                    activity.getActId(),
                    activity.getCost(),
                    activity.getDuration(),
                    activity.getEarliestStart(),
                    activity.getLatestStart(),
                    activity.getEarliestFinish(),
                    activity.getLatestFinish(),
                    dependenciesStr
            ));
        }

        System.out.println("Schedule exported successfully to: " + file.getAbsolutePath());

    } catch (IOException e) {
        System.out.println("Error while exporting schedule: " + e.getMessage());
    }
}
```

Time Complexity `exportScheduleToCSV()`: O(V + E)

- It calls `getPert_CPMWithoutDuration()`, which has a time complexity of **O(V + E).**
- The outer loop over activities has a time complexity of **O(V).**
- For each activity, in the worst case, `incomingEdges` iterates over all vertices and performs a lookup for each vertex. This results in **O(V)** for each call. Since `incomingEdges` is called once for each activity, this step contributes **O(V^2)**.
- The reduce operation on dependencies has a time complexity of **O(P),** where P is the number of dependencies per activity, related to E.
- Writing the data to the file involves **O(V)** operations.
- Combining all these, the final complexity is **O(V + E).**

```java
/**
 * Builds the PERT/CPM graph without duration.
 */
public MapGraph<String, String> getPert_CPMWithoutDuration() {
    MapGraph<String, String> pert_CPM_without_duration = new MapGraph<>(true);
    for (String vertex : pert_CPM.vertices()) {
        pert_CPM_without_duration.addVertex(vertex.split(" ")[0]);
    }
    for (String vertex : pert_CPM.vertices()) {
        for (Edge<String, String> edge : pert_CPM.outgoingEdges(vertex)) {
            pert_CPM_without_duration.addEdge(vertex.split(" ")[0], edge.getVDest().split(" ")[0], "0");
        }
    }
    return pert_CPM_without_duration;
}
```

Time Complexity `getPert_CPMWithoutDuration()`: O(V + E)

- The first loop iterates through all vertices in pert_CPM, which takes **O(V),** where V is the number of vertices.
- The second loop iterates through all vertices again, and for each vertex, it iterates through its outgoing edges. The complexity of this part is **O(E),** where E is the number of edges.
- The split operation and the `addVertex` and `addEdge` methods each run in constant time **O(1).**
- Combining these steps, the overall time complexity is **O(V + E),** where V is the number of vertices and E is the number of edges in pert_CPM.

USEI22

findCriticalPaths()

- **Method Calls**:
    - Iterates over all predecessors of the "END" activity: **O(n)**, where n is the number of activities.
    - Calls findCriticalPathsRec(activity, path, criticalPaths) recursively: **O(m+c·l)**, where m is the number of edges, c is the number of critical paths, and l is the average length of a critical path.
    - Reverses all critical paths: **O(c·l).**
- **Complexity**: The total complexity is determined by the recursive calls and the reversal of critical paths.
  Combining everything, the overall complexity is **O(n+m+c·l)** so it has an overall complexity of **O(c·l).**

```java
public LinkedHashMap<Integer, List<Activity>> findCriticalPaths() {
    LinkedHashMap<Integer, List<Activity>> criticalPaths = new
```

```
LinkedHashMap<>();
    List<Activity> path = new ArrayList<>();
    // The method will start from the "END" node
    Activity endActivity = activitiesPERT_CPM.get("END");
    // We will need to get the predecessors of the "END" node
    List<String> endPredecessors =
activitiesPERT_CPM.get("END").getPrevActIds();
    for (String endPredecessor : endPredecessors) {
        Activity activity = activitiesPERT_CPM.get(endPredecessor);
        path.add(endActivity);
        path.add(activity);
        findCriticalPathsRec(activity, path, criticalPaths);
        path.clear();
    }
    // The list will be inverted so it goes from start to end
    for (List<Activity> criticalPath : criticalPaths.values()) {
        Collections.reverse(criticalPath);
    }
    return criticalPaths;
}
```

findCriticalPathsRec(Activity activity, List<Activity> path, LinkedHashMap<Integer, List<Activity>> criticalPaths)

- **Steps**:
    - **Check Slack**: If slack is not zero, return immediately. **O(1)**.
    - **Base Case ("START" Node)**: Copy the path and add it to criticalPaths. Copying the path is **O(l)**, where **l** is the path length.
    - **Recursive Calls for Predecessors**:
        - Iterates over the predecessor activities of the current activity: **O(n)**.
        - For each predecessor:
            - Adds it to path: **O(1)**.
            - Calls findCriticalPathsRec: Recursive cost **O(m)**.
            - Removes the predecessor from path: **O(1)**.
- **Complexity**:
    - Each edge in the graph is processed once during recursion, so traversing all edges contributes **O(m)**.
    - Path copying for ccc critical paths contributes **O(c·l)**.
- **Overall Complexity**: **O(m+c·l)**

```
private void findCriticalPathsRec(Activity activity, List<Activity> path,
LinkedHashMap<Integer, List<Activity>> criticalPaths) {
```

```
    if (activity.getSlack() != 0) {
        return;
    }
    if (activity.getActId().equals("START")) {
        List<Activity> pathCopy = new ArrayList<>(path);
        criticalPaths.put(criticalPaths.size() + 1, pathCopy);
        return;
    }
    List<String> predecessors = activity.getPrevActIds();
    for (String predecessor : predecessors) {
        Activity predecessorActivity = activitiesPERT_CPM.get(predecessor);
        path.add(predecessorActivity);
        findCriticalPathsRec(predecessorActivity, path, criticalPaths);
        path.remove(predecessorActivity);
    }
}
```

USEI23

```
public List<Activity> getBottleneckActivities() {
  Map<Activity, Integer> dependencyCount = new HashMap<>();
  Map<Activity, Integer> pathCount = new HashMap<>();
  // Initialize counts
  for (Activity activity : activities.values()) {
    dependencyCount.put(activity, 0);
    pathCount.put(activity, 0);
  }

  // Count dependencies
  for (Activity activity : activities.values()) {
    for (String prevActId : activity.getPrevActIds()) {
      Activity prevActivity = activities.get(prevActId);
      if (prevActivity != null) {
        dependencyCount.put(prevActivity, dependencyCount.get(prevActivity) + 1);
      }
    }
  }

  // Count paths
  LinkedHashMap<Integer, List<Activity>> criticalPaths = findCriticalPaths();
  for (List<Activity> path : criticalPaths.values()) {
    for (Activity activity : path) {
      pathCount.put(activity, pathCount.getOrDefault(activity, 0) + 1);
    }
  }

  // Find maximum counts
  int maxDependencies = Collections.max(dependencyCount.values());
  int maxPaths = Collections.max(pathCount.values());



  // Collect bottleneck activities
```

```
    List<Activity> bottleneckActivities = new ArrayList<>();
    for (Activity activity : activities.values()) {
        if (!activity.getActId().equals("START") && !activity.getActId().equals("END")) {
            if ((dependencyCount.get(activity) > 1 && dependencyCount.get(activity) == maxDependencies) ||
(pathCount.get(activity) == maxPaths && dependencyCount.get(activity) > 1)) {
                bottleneckActivities.add(activity);
            }
        }
    }

    // Identify predecessors of the "END" node
    List<String> endPredecessors = activitiesPERT_CPM.get("END").getPrevActIds();
    Activity mostCriticalPredecessor = null;
    int maxPredecessorPaths = 0;

    for (String endPredecessor : endPredecessors) {
        Activity activity = activitiesPERT_CPM.get(endPredecessor);
        int pathsThroughPredecessor = pathCount.getOrDefault(activity, 0);
        if (pathsThroughPredecessor > maxPredecessorPaths) {
            maxPredecessorPaths = pathsThroughPredecessor;
            mostCriticalPredecessor = activity;
        }
    }

    // Add the most critical predecessor to the bottleneck activities list
    if (mostCriticalPredecessor != null && !bottleneckActivities.contains(mostCriticalPredecessor)) {
        bottleneckActivities.add(mostCriticalPredecessor);
    }

    return bottleneckActivities;
}
```

getBottleneckActivities():

- **Steps**:
  - Initializing dependencyCount and pathCount: **O(n)** if there are n activities.
  - Counting dependencies: If each activity as m predecessors, the complexity of the loop is O(n*m), where m is the average number of predecessors per activity, in the worst case m equals n so the complexity will be **$O(n^2)$**.
  - Finding critical paths: **O(c*l)**, as explained before, but in the worst case, there will be **O(n)** paths, each of length **O(n)** so the complexity will be **$O(n^2)$**.
  - Finding maximum dependency and path counts: Both uses of Collections.max have a complexity of O(n) so the complexity will be O(2n) which equals **O(n)**.

- Collecting the bottleneck activities: Loops over the activities.values() which is O(n) and performs constant-time checks and conditions for each activity, so this part has a time complexity of **O(n)**.
- Identifying predecessors of the “End” node: Loop over the endPredecessors list, the complexity is O(k), k being the number of predecessors of the node “End”, since k will likely be much smaller than n it can be considered **O(1).**
- Adding the Most Critical Predecessor: Checking if the mostCriticalPredecessor is already in the bottleneckActivities list and adding it if it's not. This step has a time complexity of **O(1)**.

- **Complexity:**
  - The final complexity of the whole method in the worst case scenario will be **O(n2)+O(n2)+O(n)=O(n2).**

USEI24

```
public void simulateDelaysAndRecalculate(LinkedHashMap<String, Integer> delays) {
  ActivitiesMapRepository activitiesMapRepository = Instances.getInstance().getActivitiesMapRepository();
  // Apply delays to specified activities
  for (String actId : delays.keySet()) {
    Activity activity_PERT_CPM = activitiesPERT_CPM.get(actId);
    Activity activity = activities.get(actId);
    if (activity != null) {
      int newDuration = activity.getDuration() + delays.get(actId);
      if (newDuration > 0) {
        activity.setDuration(newDuration);
        activity_PERT_CPM.setDuration(newDuration);
      }else if(newDuration < 0){
        throw new IllegalArgumentException("Activity duration must be positive or zero.\n" +
            "Activity ID: " + actId + "\n" + "Current Duration: " + activity.getDuration() + "\n");
      }
    }
  }

  // Recalculate times
  CalculateTimes calculateTimes = new CalculateTimes();
  calculateTimes.calculateTimes();

  // Recalculate Slack times
  for (Activity activity : activities.values()) {
    activity.calculateSlack();
  }
  activitiesMapRepository.setActivitiesMapRepository(activities);
}

// Method to calculate total project duration
```

```java
public String calculateTotalProjectDuration() {
    double maxFinishTime = 0.0;
    String unit = activities.values().iterator().next().getDurationUnit();
    for (Activity activity : activities.values()) {
        if (activity.getEarliestFinish() > maxFinishTime) {
            maxFinishTime = activity.getEarliestFinish();
        }
    }

    if (maxFinishTime > 1 && !unit.endsWith("s")) {
        unit += "s";
    }

    return maxFinishTime + " " + unit;
}
```

simulateDelaysAndRecalculate(LinkedHashMap<String, Integer> delays)

- Steps:
  - Apply Delays to Specified Activities: The method iterates over the keys in the delays map (which has d keys, where d is the number of delayed activities). For each delayed activity, it adjusts the duration. Looping through d activities gives **O(d)**.
  - Recalculate times: As explained before the complexity of calculateTimes() is **$O(n^3)$.**
  - Recalculate Slack Times: Iterating over all activities to recalculate slack times takes **O(n).**

- **Complexity:** $O(d)+O(n^3)+O(n)+O(n)$, as $O(n^3)$ dominates over the rest the complexity will be $O(n^3)$

CalculateProjectDuration()

- Iterating over all activities to find the maximum finish time takes **O(n)**