COMPLEXITY

US01

```
public static Map<Integer, Item> readItems(String fileName) throws
FileNotFoundException {
    Map<Integer, Item> items = new HashMap<>();
    int increment = 1;


    InputStream inputStream =
InputFileReader.class.getResourceAsStream("/" + fileName);
    if (inputStream == null) {
        throw new FileNotFoundException("File not found: " +
fileName);
    }

    try (Scanner scanner = new Scanner(inputStream)) {
        scanner.nextLine();

        while (scanner.hasNextLine()) {
            String[] data = scanner.nextLine().split(";");
            int id = Integer.parseInt(data[0]);
            String priorityStr = data[1].trim();
            Priority priority = Priority.fromString(priorityStr);
            List<String> operations = new
ArrayList<>(Arrays.asList(data).subList(2, data.length));
            Item item = new Item(id, priority, new ArrayList<>());
            item.getOperations().addAll(operations);
            items.put(increment, item);
            increment++;
        }
    }

    return items;
}
```

- **while (scanner.hasNextLine()): O(n * m)**
  The loop iterates **n** times, where **n** is the number of lines in the file (or items). Each iteration processes a line, which involves reading and splitting the line into components (ID, priority, and operations). Assuming **m** is the average length of a line, the time complexity per iteration is **O(m)**. Therefore, the overall complexity for the loop is **O(n * m)**.

```
public static Map<Integer, Workstation> readMachines(String fileName)
throws FileNotFoundException {
    Map<Integer, Workstation> machines = new HashMap<>();
    int increment = 1;


    InputStream inputStream =
InputFileReader.class.getResourceAsStream("/" + fileName);
    if (inputStream == null) {
        throw new FileNotFoundException("File not found: " +
fileName);
    }

    try (Scanner scanner = new Scanner(inputStream)) {
        scanner.nextLine();

        while (scanner.hasNextLine()) {
            String[] data = scanner.nextLine().split(";");
            String id = data[0];
            String operation = data[1];
            int time = Integer.parseInt(data[2]);
            Workstation workstation = new Workstation();
            workstation.setId(id);
            workstation.setOperation(operation);
            workstation.setTime(time);
            machines.put(increment, workstation);
            increment++;
        }
    }

    return machines;
}
```

- **while (scanner.hasNextLine()): O(n)**
  The loop iterates **n** times, where **n** is the number of lines in the file, corresponding to the number of machines. For each iteration, the line is split into three parts (ID, operation, and time), and a new Workstation object is created and added to the map. Since the operations per iteration (splitting the line and adding to the map) are constant-time operations, the overall complexity for the loop is **O(n)**.

US02

Method simulateProcessUS02

- This method will simulate the process considering the time each process takes and putting the processes that have the smallest initial time first.
- Having an overall complexity of $O(n \cdot k \cdot m^2)$.

```
public static LinkedHashMap<String, Double> simulateProcessUS02() {
    HashMap<Item, Workstation> ProdPlan =
HashMap_Items_Workstations.getProdPlan();
    HashMap<String, LinkedList<Item>> operationsQueue = new
HashMap<>();
    ArrayList<Workstation> workstations = new
ArrayList<>(ProdPlan.values());
    ArrayList<Item> items = new ArrayList<>(ProdPlan.keySet());
    removeNullMachines(workstations);
    removeNullItems(items);
    // AC1 - Create the operationsQueue with the list of the items for
each operation
```

```
    fillOperationsQueue(items, operationsQueue);
    // AC2 - Assign the items to the machines
    LinkedHashMap<String, Double> timeOperations = new
LinkedHashMap<>();
    fillUpMachinesUS02(operationsQueue, workstations, timeOperations,
items);
    for (Workstation workstation : workstations) {
        workstation.clearUpWorkstation();
    }
    for (Item item : items) {
        item.setCurrentOperationIndex(0);
        item.setLowestTimes(new LinkedHashMap<>());
        // Calculate and set waiting times for each operation
        for (String operation : item.getOperations()) {
            long entryTime = item.getEntryTime(operation);
            long currentTime = System.currentTimeMillis();
            int waitTime = (int) (currentTime - entryTime);
            item.setWaitingTime(operation, waitTime);
        }
    }
    return timeOperations;
}
```

- **HashMap_Items_Machines.getProdPlan(): O(n)** as we are assuming that n is the number of items, could be O(m) but because in the file provided there are more items than workstations, we are going to say this line as a complexity O(n).
- **new ArrayList<>(ProdPlan.values()): O(n)** since the values are copied.
- The workstations get cleared having a complexity of m.
- Then it will go through each of the items and get the waiting time of each operation (not part of the complexity of this simulator but it is important to make a reference), having a complexity of $\boldsymbol{O(n \cdot k)}$.
- With what was discussed below we can say that the complexity is going to be $\boldsymbol{O(n \cdot k + + n \cdot k \cdot m^2 + n + m)}$. Which can be simplified to $\boldsymbol{O(n \cdot k \cdot m^2)}$.
- **removeNullWorkstations(workstations): O(m)** where m is the number of workstations. In the method we are going to iterate through all the workstations.

```
private static void removeNullWorkstations(ArrayList<Workstation>
workstations) {
    workstations.removeIf(machine ->
machine.getId().equalsIgnoreCase(""));
}
```

- **removeNullItems(items): O(n)** where n is the number of items. In the method we are going to iterate through all the items.

```
private static void removeNullItems(ArrayList<Item> items) {
    items.removeIf(item -> item.getId() == 0);
}
```

- **fillOperationsQueue(items, operationsQueue): O$(n \cdot k)$** where k is the number of operations per item. So, we can assume that the complexity is going to be the sum of all operations that are present in each of the items. As you can see there are two loops being why we could say that he is summing n times k but we don't say it is because k can have different values depending on the circumstance.

```
private static void fillOperationsQueue(ArrayList<Item> items,
HashMap<String, LinkedList<Item>> operationsQueue) {
    for (Item item : items) {
        ArrayList<String> operations = (ArrayList<String>)
item.getOperations();
        if (operations.isEmpty()) {
            System.out.println("Item with ID " + item.getId() + " has
no operations.");
            continue;
        }
        for (String operation : operations) {
            if (!operationsQueue.containsKey(operation)) {
                operationsQueue.put(operation, new LinkedList<>());
            }
            operationsQueue.get(operation).add(item);
            // Set entry time for the operation
            item.setEntryTime(operation, System.currentTimeMillis());
        }
    }
}
```

Method fillUpMachinesUS02

This method is a part of the complexity of simulateProcess02 but is composed of several more methods.

- Having a complexity of $\boldsymbol{O(m\log m + n \cdot k \cdot m + n \cdot k^2 + n\log n \cdot k\log k + n \cdot k \cdot m^2)}$. Considering the methods listed below. $\boldsymbol{O(n \cdot k \cdot m^2)}$ being the worst-case scenario.

```
private static void fillUpMachinesUS02(HashMap<String,
LinkedList<Item>> operationsQueue, ArrayList<Workstation>
workstations, LinkedHashMap<String, Double> timeOperations,
ArrayList<Item> items) {
    int quantMachines = workstations.size();
    sortMachinesByTime(workstations);
    sortItemsByTime(items, workstations);
    addAllItems(operationsQueue, workstations, timeOperations, items,
quantMachines);
}
```

- **int quantMachines = workstations.size():** is *O(1)* quantMachines is equal to the size of the workstations list. And size() as a complexity of 1.
- **sortMachinesByTime(workstations):** is $O(m\log m)$ because it will get the time of each workstation and compare them between each other, so it goes m times through getTime() and sort() uses an mixture of merge and insertion sort (TimSort), so it will divide the m by 2 until it is able to sort it being why it will go through $\boldsymbol{O(2m\log m)}$ because the getTime() will go through twice to get the value of both workstations so if we simplify it the complexity is $\boldsymbol{O(m\log m)}$.

```
private static void sortMachinesByTime(ArrayList<Workstation>
workstations) {
    workstations.sort(Comparator.comparingInt(Workstation::getTime));
}
```

- **sortItemsByTime(items, workstations):** is divided into some operations so lets go through them:
- As visualized bellow the joint complexity of **addTimes() and swapOperations()** is $\boldsymbol{O(n \cdot k \cdot m + n \cdot k^2)}$.
- So now we need to consider items.sort():
- **items.sort()** will use an comparator that compares all the items in the system based on the list of their lowest times, that were determined before in **addTimes()**, so we can easily find what are the overall fastest items to complete the process and put them in the list correctly sorted by overall time to complete as the list is sorted by time. Now in terms of complexity we have a sort of the items where it is going to sort two different items by the time they take by each of their operations lowest times, and because it uses TimSort (sort method was explained before), so it will have a complexity of $\boldsymbol{O(n\log n)}$ we ignore the fact they do it twice because the complexity stays the same, then it goes into a loop where it goes h times through the loop while also inside comparing the values of both items times resulting on the use of a compare that uses sort wish leads to the complexity of $\boldsymbol{O(k\log k)}$. So the sort has an overall complexity $\boldsymbol{O(n\log n \cdot k\log k)}$
- Meaning that the overall complexity of **sortItemsByTime()** is $\boldsymbol{O(n \cdot k \cdot m + n \cdot k^2 + n\log n \cdot k\log k)}$.

```
private static void sortItemsByTime(ArrayList<Item> items,
ArrayList<Workstation> workstations) {
    addTimes(items, workstations);
    swapOperations(items);
    items.sort((item1, item2) -> {
        LinkedHashMap<String, Integer> sortedTimes1 =
item1.getLowestTimes().entrySet().stream()
                .sorted(Map.Entry.comparingByValue())
                .collect(LinkedHashMap::new, (map, entry) ->
map.put(entry.getKey(), entry.getValue()), LinkedHashMap::putAll);

        LinkedHashMap<String, Integer> sortedTimes2 =
item2.getLowestTimes().entrySet().stream()
                .sorted(Map.Entry.comparingByValue())
                .collect(LinkedHashMap::new, (map, entry) ->
map.put(entry.getKey(), entry.getValue()), LinkedHashMap::putAll);

        Iterator<Map.Entry<String, Integer>> it1 =
sortedTimes1.entrySet().iterator();
        Iterator<Map.Entry<String, Integer>> it2 =
sortedTimes2.entrySet().iterator();

        while (it1.hasNext() && it2.hasNext()) {
            Map.Entry<String, Integer> entry1 = it1.next();
            Map.Entry<String, Integer> entry2 = it2.next();
            int timeComparison = Integer.compare(entry1.getValue(),
entry2.getValue());
            if (timeComparison != 0) {
                return timeComparison;
            }
        }
        return 0;
    });
}
```

- **addTimes(items, workstations):** is ***O(n·k·m)*** assuming n is for items, k for operations and m for workstations as said before. That is going to be the complexity because it will iterate over the items n times then it will iterate over each item operations k times and then it will go through all the machines m times to get the smallest time of each operation that the item has. So, there are three loops for each of the objects considered that are inside each other which means that the complexity is O(n·k·m).

```
private static void addTimes(ArrayList<Item> items,
ArrayList<Workstation> workstations) {
    for (Item item : items) {
        LinkedHashMap<String, Integer> operationTimes = new
LinkedHashMap<>();
        for (String operation : item.getOperations()) {
            int minTime = Integer.MAX_VALUE;
            for (Workstation workstation : workstations) {
                if
(workstation.getOperation().equalsIgnoreCase(operation)) {
                    minTime = Math.min(minTime,
workstation.getTime());
                }
            }
            if (minTime == Integer.MAX_VALUE) {
                System.out.println("Warning: No workstation found for
operation: " + operation);
            }
            operationTimes.put(operation, minTime);
        }
        item.setLowestTimes(operationTimes);
    }
}
```

- **swapOperations(items):** is $\boldsymbol{O(n \cdot k^2)}$ assuming n items and k operations. The program will go through each item n times and then will go through each item operations k^2 times and compare each of those operations until every single operation will be eventually swapped until it is sorted, this could be compared in some way to bubble sort but with a different way of sorting. So the worst case will be **O(n $\cdot$ k^2).**

```
private static void swapOperations(ArrayList<Item> items) {
    for (Item item : items) {
        List<String> operations = item.getOperations();
        LinkedHashMap<String, Integer> lowestTimes =
item.getLowestTimes();
        boolean swapped;
        do {
            swapped = false;
            for (int i = 0; i < operations.size() - 1; i++) {
                String operation1 = operations.get(i);
                String operation2 = operations.get(i + 1);
                if (lowestTimes.get(operation1) >
lowestTimes.get(operation2)) {
                    Collections.swap(operations, i, i + 1);
                    swapped = true;
                }
            }
        } while (swapped);
    }
}
```

Method addAllItems

- Before continuing the method will go through the items n times and create a map with the quantity of each item produced this will be helpful to distinguish items with the same id. The method has a complexity of **O(n).**
- This method will go through all the items n times and entering the addOperations n times. Having the complexity of **O(n) times the complexity of the rest of the program which continues in addOperations().**
- So the complexity is $\boldsymbol{O(n \cdot k \cdot m^2)}$. For this method.

```
private static void addAllItems(HashMap<String, LinkedList<Item>>
operationsQueue, ArrayList<Workstation> workstations,
LinkedHashMap<String, Double> timeOperations, ArrayList<Item> items,
int quantMachines) {
    LinkedHashMap<String, Integer> quantItems = new LinkedHashMap<>();
    for (Item item : items) {
        quantItems.put(String.valueOf(item.getId()), 0);
    }
    for (Item item : items) {
        quantItems.put(String.valueOf(item.getId()),
quantItems.get(String.valueOf(item.getId())) + 1);
        if (item.getId() != 0) {
            quantMachines = addOperations(operationsQueue,
workstations, timeOperations, item, quantMachines, quantItems);
        }
    }
}
```

Method addOperations

- This method has a complexity of $O(k \cdot m^2)$ because of the loop that goes through all the operations of a certain item k times and then will go through all the operations and check the ones that have that operation.
- Then it will call **checkMachines()** if the quantity of machines is 0, having a complexity of m as it goes through m times and clear the machines.
- It will also call **checkMachinesWithOperations()** if the quantity is not 0, having a complexity of m as it is explained below.
- And the last method called is **addItem()** that as a complexity of m as it will go through all available items.

```
private static int addOperations(HashMap<String, LinkedList<Item>>
operationsQueue, ArrayList<Workstation> workstations,
LinkedHashMap<String, Double> timeOperations, Item item, int
quantMachines, LinkedHashMap<String, Integer> quantItems) {
    for (String operation : item.getOperations()) {
        ArrayList<Workstation> availableWorkstations = new
ArrayList<>();
        for (Workstation workstation : workstations) {
            if
(workstation.getOperation().equalsIgnoreCase(operation)) {
                availableWorkstations.add(workstation);
            }
        }
        if (quantMachines == 0) {
            quantMachines = checkMachines(workstations,
quantMachines);
        } else {
            quantMachines =
checkMachinesWithOperation(availableWorkstations, quantMachines,
operation);
        }
        quantMachines = addItem(operationsQueue, timeOperations, item,
operation, availableWorkstations, quantMachines, quantItems);
    }
    return quantMachines;
}
```

- The method will check if all the workstations are filled up if yes then it will clear them up this is so there are always machines available in the extreme cases, being why the complexity is just m as it just goes to all the workstations and clears them.

```java
private static int checkMachines(ArrayList<Workstation> workstations, 
int quantMachines) {

    for (Workstation workstation1 : workstations) {
        workstation1.clearUpWorkstation();
    }
    quantMachines = workstations.size();

    return quantMachines;
}
```

- This method does a similar process to checkMachines but now will just clear up the machines of a certain operation if they are all filled up.
- Because of the similarity it has the same complexity of m, but does two different things being them creating a list of the machines that have that operation and have an item and also seeing if any of those machines still have an item so it doesn't clear else it will clear.

```java
private static int checkMachinesWithOperation(ArrayList<Workstation> 
workstations, int quantMachines, String operation) {
    boolean notFree = true;
    int quant = 0;
    ArrayList<Workstation> tempMachines = new ArrayList<>();
    for (Workstation workstation : workstations) {
        if (workstation.getOperation().contains(operation) && 
workstation.getHasItem()) {
            quant++;
            tempMachines.add(workstation);
        }
        if (workstation.getOperation().equalsIgnoreCase(operation) && 
!workstation.getHasItem()) {
            notFree = false;
        }
    }
    if (quant >= 1 && notFree) {
        for (Workstation workstation : tempMachines) {
            workstation.clearUpWorkstation();
        }
        quantMachines = quantMachines + quant;
    }
    return quantMachines;
}
```

- This method has a complexity of **m** because it goes through all the available machines and add the item operation to a map that has the information of that operation and the time it took to be completed.

```java
private static int addItem(HashMap<String, LinkedList<Item>> 
operationsQueue, LinkedHashMap<String, Double> timeOperations, Item 
item, String operation, ArrayList<Workstation> availableWorkstations, 
int quantMachines, LinkedHashMap<String, Integer> quantItems) {
    for (Workstation workstation : availableWorkstations) {
        if (item.getCurrentOperationIndex() >
item.getOperations().size()) {
                return quantMachines;
            }
            if
((operationsQueue.get(workstation.getOperation()).contains(item) &&
workstation.getOperation().equalsIgnoreCase(operation)) &&
(!workstation.getHasItem())) {
                int currentItem = timeOperations.size() + 1;
                workstation.setHasItem(true);

item.setCurrentOperationIndex(item.getCurrentOperationIndex() + 1);
                quantMachines--;
                String operation1 = currentItem + " - " + " Operation: " +
operation + " - Machine: " + workstation.getId() + " - Priority: " +
item.getPriority() + " - Item: " + item.getId() + " - Time: " +
workstation.getTime() + " - Quantity: " +
quantItems.get(String.valueOf(item.getId()));
                timeOperations.put(operation1,
timeOperations.getOrDefault(workstation.getOperation(), 0.0) +
workstation.getTime());

operationsQueue.get(workstation.getOperation()).remove(item);
                return quantMachines;
            }
        }
        return quantMachines;
}
```

US03

```
public static TreeMap<Item, Double>
calculateTotalProductionTimePerItem() {
    HashMap<Item, Double> totalProductionTimePerItem = new
HashMap<>();
    LinkedHashMap<String, Double> timeOperations =
simulateProcessUS02();

    HashMap<Item, Workstation> ProdPlan =
HashMap_Items_Workstations.getProdPlan();
    ArrayList<Item> items = new ArrayList<>(ProdPlan.keySet());

    for (Item item : items) {
        double totalProductionTime = 0.0;
        for (String operation : item.getOperations()) {
            for (Map.Entry<String, Double> entry :
timeOperations.entrySet()) {
                if (entry.getKey().contains(operation)) {
                    totalProductionTime += entry.getValue();
                }
            }
        }
        totalProductionTimePerItem.put(item, totalProductionTime);
    }
    return sortById(removeDuplicateItems(totalProductionTimePerItem));
}
```

- **for (Item item : items): O(n)**, where n is the number of items.

- **for (String operation : item.getOperations()): O(m),** where m is the number of operations per item. This loop is nested inside the previous loop.
- **for (Map.Entry<String, Double> entry : timeOperations.entrySet()): O(k)**, where k is the number of entries in timeOperations. This loop is nested inside the operations loop.
- **Thus, the total complexity of the method is: O(n*m*k).**

```java
public static HashMap<Item, Double> removeDuplicateItems(HashMap<Item,
Double> totalProductionTimePerItem) {
    HashMap<Item, Double> uniqueTotalProductionTimePerItem = new
HashMap<>();
    Set<Integer> uniqueIds = new HashSet<>();

    for (Map.Entry<Item, Double> entry :
totalProductionTimePerItem.entrySet()) {
        Item item = entry.getKey();
        int itemId = item.getId();

        if (!uniqueIds.contains(itemId)) {
            uniqueTotalProductionTimePerItem.put(item,
entry.getValue());
            uniqueIds.add(itemId);
        }
    }

    return uniqueTotalProductionTimePerItem;
}
```

- **for (Map.Entry<Item, Double> entry totalProductionTimePerItem.entrySet()): O(n)**, where n is the number of items in the totalProductionTimePerItem.

```java
public static TreeMap<Item, Double> sortById(HashMap<Item, Double>
totalProductionTimePerItem) {
    TreeMap<Item, Double> sortedMap = new
TreeMap<>(Comparator.comparingInt(Item::getId));
    sortedMap.putAll(totalProductionTimePerItem);
    return sortedMap;
}
```

- **sortedMap.putAll(totalProductionTimePerItem): O (m log m),** to insert m elements into a TreeMap, the complexity is O(m log m).

US04

```java
public HashMap<String, Double> calcOpTime(LinkedHashMap<String, Double> timeOperations) {
  try {
    HashMap<String, Double> OpTime = new HashMap<>();
    for (String operation : timeOperations.keySet()) {
      String[] parts = operation.split(" - ");
      String operationName = parts[1].split(": ")[1];
      OpTime.putIfAbsent(operationName, 0.0);
      OpTime.put(operationName, OpTime.get(operationName) + timeOperations.get(operation));
    }
    return OpTime;
  } catch (Exception e) {
    System.out.println("Error: " + e.getMessage());
  }
  return null;
}
```

- for(String operations : timeOperations.keyset()): ***O(n)***, where n is the number of keys in timeOperations.
- String[] parts = operation.split(" - ") and String operationName = parts[1].split(": ")[1]: O(1) for each iteration, because the length of the string is relatively small. Since the splits are executed every iteration their contributions is ***O(n)x O(1) = O(n).***
- OpTime.putIfAbsent() and OpTime.put(): ***O(1)***, as operations put, putIfAbsent and get have O(1) complexity on a HashMap. Since we perform a constant number of operations for each of the n entries in timeOperations, this part also contributes ***O(n)*** in total.
- In total the complexity of this method is ***O(n)*.**

US05

```
public void listWorkstationsByAscOrder(LinkedHashMap<String, Double> timeOperations) {
    int totalExecutionTime = 0;

    // Calculate total execution time
    for (Map.Entry<String, Double> entry : timeOperations.entrySet()) {
      Double time = entry.getValue();
      totalExecutionTime += time;
    }

    // Store workstations with total time
    HashMap<String, Double> workstations = new HashMap<>();
    for (String workstation : timeOperations.keySet()) {
      double timeWkStation = 0;
      Workstation ws = new Workstation();
      String[] parts = workstation.split(" - ");
      String workstationName = parts[2].split(": ")[1];
      timeWkStation += timeOperations.get(workstation);
      ws.setId(workstationName);
      if (workstations.containsKey(workstationName)) {
        workstations.put(workstationName, workstations.get(workstationName) + timeWkStation);
      } else {
        workstations.put(workstationName, timeWkStation);
      }
    }
// Calculate the percentage of total time for each workstation
    HashMap<String, Double> workstationPercentages = new HashMap<>();
    for (Map.Entry<String, Double> entry : workstations.entrySet()) {
      double time = entry.getValue();
      double percentage = time / totalExecutionTime * 100;
      workstationPercentages.put(entry.getKey(), percentage);
    }

// Print sorted workstations with total time and percentage
    workstationPercentages.entrySet().stream()
        .sorted(Map.Entry.comparingByValue())
        .forEach(entry -> System.out.printf("%s - Total time: %.0f - Percentage: %.2f%%\n",
entry.getKey(), workstations.get(entry.getKey()), entry.getValue()));
  }
```

- for (Map.Entry<String, Double> entry : timeOperations.entrySet()): ***O(n)*** since the loop iterates for n entries.
- for (String workstation : timeOperations.keySet()): ***O(n)*** since the splits inside are ***O(1)*** and the containsKey, put and get operations on a HashMap are ***O(1)*** but their used for each iteration making it ***O(n)***.
- for (Map.Entry<String, Double> entry : workstations.entrySet()): The operations inside the loop are all ***O(1)***, as get and put are ***O(1)***, since the loop is executed n times the complexity will be ***O(n)***.
- workstationPercentages.entrySet().stream() .sorted(Map.Entry.comparingByValue()) .forEach(entry -> System.out.printf(...)): Sorting is a ***O(m*log m)***, where m is the number of workstations. In the worst case can be as large as n, so sorting is ***O(n*log n)***. The forEach loop prints each entry, so it is ***O(n)***. The complexity of this part is ***O(n*log n)***.
- The most significant term is ***O(n*log n)***, which dominates the overall complexity of the method. So the method is ***O(n*log n)***.

US06

Method caculateAvgExecutionWaitingTimes

- Returns HashMap<String, Double[]> where:

- Key: Operation name

 - Value: Array containing [avg execution time, avg waiting time]

- Process: 1. Groups items by operation

2. Finds fastest machine for each operation

3. Calculates execution time based on machine speed

4. Calculates waiting time based on queue position

5. Computes averages for both metrics

- Time Complexity: O(o * m * i), where:
- o is the number of operations
- m is the number of machines
- i is the number of items in each operation queue
- Breakdown:
- Operations queue iteration: O(o)
- For each operation:
  - Finding fastest machine: O(m)
  - Processing items in queue: O(i)
- The nested structure makes it O(o * m * i)

```java
public static HashMap<String, Double[]> calculateAvgExecutionAndWaitingTimes() {
    HashMap<String, Double[]> operationTimes = new HashMap<>();
    HashMap<Item, Workstation> ProdPlan =
Instances.getInstance().getHashMapItemsWorkstations().getProdPlan();
    LinkedHashMap<String, Double> timeOperations = simulateProcessUS02();
    HashMap<String, LinkedList<Item>> operationsQueue = new HashMap<>();
    ArrayList<Workstation> machines = new ArrayList<>(ProdPlan.values());
    removeNullMachines(machines);
    removeNullItems(ProdPlan);


    // Fill the operationsQueue with items waiting for each operation
    fillOperationsQueueus06(ProdPlan, operationsQueue);


    // Track execution and waiting times for each operation
    for (String operation : operationsQueue.keySet()) {
        double totalExecutionTime = 0.0;
        double totalWaitingTime = 0.0;
        int itemCount = 0;
            ArrayList<Double> executionTimes = new ArrayList<>();
        // Calculate waiting time for each item in the queue
        for (Map.Entry<String, Double> entry : timeOperations.entrySet()) {
            String[] parts = entry.getKey().split(" - ");
            String operationName = parts[1].split(": ")[1];
            if (operation.equals(operationName)) {
                if (executionTimes.isEmpty()){
                    executionTimes.add(entry.getValue());
                } else {
                    executionTimes.add(entry.getValue() +
executionTimes.get(executionTimes.size() - 1));
                }
                totalExecutionTime += entry.getValue();
                executionTimes.add(entry.getValue());
                for (Double time : executionTimes) {
                    totalWaitingTime += time;
                }
                itemCount++;
            }
        }
        // Calculate averages
        double avgExecutionTime = itemCount > 0 ? totalExecutionTime / itemCount : 0;
        double avgWaitingTime = itemCount > 0 ? totalWaitingTime / itemCount : 0;
        operationTimes.put(operation, new Double[]{avgExecutionTime, avgWaitingTime});
    }

    return operationTimes;
}
```

Method fillOperationsQueueus06

- Input: - ProdPlan: HashMap<Item, Workstation> - Maps items to their assigned workstations - operationsQueue: HashMap<String, LinkedList<Item>> - Will hold operation queues

- Process: For each item in the production plan: - Gets list of operations required for the item - For each operation: - Creates a new queue if operation doesn't exist - Adds item to operation's queue.

- O(i * o), where:
- i = number of items in ProdPlan
- o = average number of operations per item
- Breakdown:
  - Outer loop through items: O(i)

  - Inner loop through operations: O(o) -

  -Queue operations (add/contains): O(1)

```java
private static void fillOperationsQueueus06(HashMap<Item, Workstation> ProdPlan, HashMap<String,
LinkedList<Item>> operationsQueue) {
    for (Item item : ProdPlan.keySet()) {
        ArrayList<String> operations = (ArrayList<String>) item.getOperations();
        for (String operation : operations) {
            if (!operationsQueue.containsKey(operation)) {
                operationsQueue.put(operation, new LinkedList<>());
            }
            operationsQueue.get(operation).add(item);
        }
    }
}
```

US07

Method generateWokstationFlowDepency

- Initial Setup: The code sets up several collections (ProdPlan, timeOperations, operationsQueue, etc.) and fills them based on other data sources.
- Time Complexity: This part mostly depends on the number of items in ProdPlan, so it’s O(n), where nis the number of items.
- Removing Null Elements: The method removes any null elements from machines and ProdPlan.
- Time Complexity: This is O(n) as it goes through all items in ProdPlan and machines once to filter them.
- Filling the operationsQueue: This populates the operationsQueue by going through ProdPlan.
- Time Complexity: O(n) as it processes each item in ProdPlan.
- Main Loop (Processing Operations): Here, the method goes through each operation in operationsQueue. For each operation, it then loops through timeOperations to check times for each item related to that operation.
- Let’s say there are k operations and pentries in timeOperations.
- Time Complexity: This part takes O(k·p) since for each operation, it goes through all timeOperations.
- Adding Results to operationTimes: At the end, each operation is added to operationTimes.
- Time Complexity: This is O(k)since there are k operations.
- Total Complexity
- Adding it all up, we get:
- O(n)+O(n)+O(n)+O(k·p)+O(k)
- O(n+k·p)

```java
public static HashMap<String, List<Map.Entry<String, Integer>>>
generateWorkstationFlowDependency() {
    HashMap<String, List<Map.Entry<String, Integer>>> flowDependency = new HashMap<>();
    HashMap<Item, Workstation> prodPlan =
Instances.getInstance().getHashMapItemsWorkstations().getProdPlan();
    ArrayList<Workstation> machines = new ArrayList<>(prodPlan.values());


    // Remove null entries and initialize flowDependency
    removeNullMachines(machines);
    removeNullItems(prodPlan);
    LinkedHashMap<String, Double> timeOperations = simulateProcessUS02();


    for (Workstation machine : machines) {
        flowDependency.put(machine.getId(), new ArrayList<>());
    }


    List<String> entries = new ArrayList<>(timeOperations.keySet());
    ArrayList<String> itemsID = timeOperations.keySet().stream()
            .map(entry -> entry.split(" - ")[4].split(": ")[1])
            .collect(Collectors.toCollection(ArrayList::new));
    for (int i = 0; i < entries.size() - 1; i++) {
        String currentEntry = entries.get(i);
        String nextEntry = entries.get(i + 1);
        String currentItemID = itemsID.get(i);
        String nextItemID = itemsID.get(i + 1);


        String[] currentParts = currentEntry.split(" - ");
        String currentMachineId = currentParts[2].split(": ")[1];


        String[] nextParts = nextEntry.split(" - ");
        String nextMachineId = nextParts[2].split(": ")[1];


        Workstation fromWorkstation = machines.stream()
                .filter(machine -> machine.getId().equalsIgnoreCase(currentMachineId))
                .findFirst()
                .orElse(null);


         Workstation toMachine = machines.stream()
                .filter(machine -> machine.getId().equalsIgnoreCase(nextMachineId))
                .findFirst()
                .orElse(null);


        if (fromWorkstation != null && toMachine != null && currentItemID.equals(nextItemID)) {
            updateTransitions(flowDependency, fromWorkstation.getId(), toMachine.getId());
        }
    }


    return sortWorkstationsByTransitions(flowDependency);
}
```

Method updateTransitions

- Input: - Flow dependency map

- Source workstation

- Destination workstation

- Process: 1. Checks if transition already exists

2. If exists: Increments counter

3. If new: Adds new transition with count 1

- Time Complexity: O(t), where:
- t is the number of existing transitions for the source machine
- Breakdown:
- Linear search through existing transitions: O(t)
- Update or add operation: O(1)

```
private static void updateTransitions(HashMap<String, List<Map.Entry<String, Integer>>>
flowDependency, String fromMachine, String toMachine) {
    // Check if the transition to `toMachine` already exists
    boolean found = false;
    for (Map.Entry<String, Integer> entry : flowDependency.get(fromMachine)) {
        if (entry.getKey().equals(toMachine)) {
            // Increment the existing transition count
            entry.setValue(entry.getValue() + 1);
            found = true;
            break;
        }
    }
    // If the transition to `toMachine` was not found, add a new entry with a count of 1
    if (!found) {
        flowDependency.get(fromMachine).add(new AbstractMap.SimpleEntry<>(toMachine, 1));
    }
}
```

Method sortWorkstationsByTransitions

- Process: 1. Calculates total transitions per workstation

2. Sorts workstations by: - Primary: Total number of transitions (descending) - Secondary: Workstation ID (ascending)

3. For each workstation, sorts its transitions by: - Primary: Transition count (descending) - Secondary: Destination workstation ID

- Time Complexity: O(w * log w + t * log t), where:
- w is the number of workstations
- t is the maximum number of transitions per workstation
- Breakdown:
- Calculating total transitions: O(w * t)
- Sorting workstations by total transitions: O(w * log w)
- Sorting transitions for each workstation: O(t * log t)
- Stream operations and sorting make this O(w * log w + t * log t)

```

private static HashMap<String, List<Map.Entry<String, Integer>>> sortWorkstationsByTransitions(
        HashMap<String, List<Map.Entry<String, Integer>>> flowDependency) {


    // Calculate total transitions for each workstation
    Map<String, Integer> totalTransitions = new HashMap<>();
    for (Map.Entry<String, List<Map.Entry<String, Integer>>> entry : flowDependency.entrySet())
{
        int total = entry.getValue().stream()
                .mapToInt(Map.Entry::getValue)
                .sum();
        totalTransitions.put(entry.getKey(), total);
    }


    // Create sorted result
    LinkedHashMap<String, List<Map.Entry<String, Integer>>> sortedFlow = new LinkedHashMap<>();


    // Sort workstations by total transitions (descending)
    totalTransitions.entrySet().stream()
```

```
            .sorted((e1, e2) -> {
                int compare = e2.getValue().compareTo(e1.getValue());
                if (compare == 0) {
                    // If transition counts are equal, sort by workstation ID
                    return e1.getKey().compareTo(e2.getKey());
                }
                return compare;
            })
            .forEach(entry -> {
                String workstationId = entry.getKey();
                List<Map.Entry<String, Integer>> transitions =
flowDependency.get(workstationId);

                // Sort transitions by count (descending) and then by destination workstation ID
                transitions.sort((a, b) -> {
                    int compareCount = b.getValue().compareTo(a.getValue());
                    if (compareCount == 0) {
                        return  a.getKey().compareTo(b.getKey());
                    }
                    return compareCount;
                });


                sortedFlow.put(workstationId, transitions);
            });


    return sortedFlow;
}
```

Method checkMachines

- Input: - List of machines - Current count of available machines

- Process: 1. If no machines available (quantMachines == 0): - Clears all machines (sets them as available) - Resets machine counter to total number

2. Returns updated machine count

```
private static int checkMachines(ArrayList<Workstation> machines, int quantMachines) {
    if (quantMachines == 0) {
        for (Workstation machine1 : machines) {
            machine1.clearUpWorkstation();
        }
        quantMachines = machines.size();
    }
    return quantMachines;
}
```

US08

Method simulateProcessUS08

- This method will simulate the process considering the time each process takes and putting the processes that have the smallest initial time first, and also considering the priority of each item meaning the priority will appear in the order high, normal and low.

```
public static LinkedHashMap<String, Double> simulateProcessUS08() {
    HashMap<Item, Workstation> ProdPlan =
HashMap_Items_Machines.getProdPlan();
    HashMap<String, LinkedList<Item>> operationsQueue = new
HashMap<>();
    ArrayList<Workstation> workstations = new
ArrayList<>(ProdPlan.values());
    ArrayList<Item> items = new ArrayList<>(ProdPlan.keySet());
    removeNullMachines(workstations);
    removeNullItems(items);
    fillOperationsQueue(items, operationsQueue);
    LinkedHashMap<String, Double> timeOperations = new
LinkedHashMap<>();
    fillUpMachinesUS08(operationsQueue, workstations, timeOperations,
items);
    for (Workstation workstation : workstations) {
        workstation.clearUpWorkstation();
    }
    for (Item item : items) {
        item.setCurrentOperationIndex(0);
        item.setLowestTimes(new LinkedHashMap<>());
    }
    return timeOperations;
}
```

- **HashMap_Items_Machines.getProdPlan(): O(n)** as we are assuming that n is the number of items, could be O(m) but because in the file provided there are more items than workstations, we are going to say this line as a complexity O(n).
- **new ArrayList<>(ProdPlan.values()): O(n)** since the values are copied.
- The workstations get cleared having a complexity of m.
- Then it will go through each of the items and get the waiting time of each operation (not part of the complexity of this simulator but it is important to make a reference), having a complexity of $\boldsymbol{O(n \cdot k)}$.
- With what was discussed below we can say that the complexity is going to be $\boldsymbol{O(n \cdot k + + n \cdot k \cdot m^2 + n + m)}$. Which can be simplified to $\boldsymbol{O(n \cdot k \cdot m^2)}$.
- **removeNullWorkstations(workstations): O(m)** where m is the number of workstations. In the method we are going to iterate through all the workstations.

```
private static void removeNullMachines(ArrayList<Workstation>
workstations) {
    workstations.removeIf(machine ->
machine.getId().equalsIgnoreCase(""));
}
```

- **removeNullItems(items): O(n)** where n is the number of items. In the method we are going to iterate trough all the items.

```
private static void removeNullItems(ArrayList<Item> items) {
    items.removeIf(item -> item.getId() == 0);
}
```

- **fillOperationsQueue(items, operationsQueue): O$(n \cdot k)$** where k is the number of operations per item. So we can assume that the complexity is going to be the sum of all operations that are present in each of the items. As you can see there are two loops being why we could say that he is summing n times k but we don't say it is because k can have different values depending on the circumstance.

```
private static void fillOperationsQueue(ArrayList<Item> items,
HashMap<String, LinkedList<Item>> operationsQueue) {
    for (Item item : items) {
        ArrayList<String> operations = (ArrayList<String>)
item.getOperations();
        if (operations.isEmpty()) {
            System.out.println("Item with ID " + item.getId() + " has
no operations.");
            continue;
        }
        for (String operation : operations) {
            if (!operationsQueue.containsKey(operation)) {
                operationsQueue.put(operation, new LinkedList<>());
            }
            operationsQueue.get(operation).add(item);
            // Set entry time for the operation
            item.setEntryTime(operation, System.currentTimeMillis());
        }
    }
}
```

Method fillUpMachinesUS08

This method is a part of the complexity of simulateProcess02 but is composed of several more methods.

- Having a complexity of $O(m \log m + n \log n + n \cdot k \cdot m + n \cdot k^2 + n \log n \cdot k \log k + n \cdot k \cdot m^2)$. Considering the methods listed below. $O(n \cdot k \cdot m^2)$ being the worst-case scenario.

```java
private static void fillUpMachinesUS08(HashMap<String,
LinkedList<Item>> operationsQueue, ArrayList<Workstation>
workstations, LinkedHashMap<String, Double> timeOperations,
ArrayList<Item> items) {
    int quantMachines = workstations.size();
    sortMachinesByTime(workstations);
    sortItemsByPriorityAndTime(items, workstations);
    addAllItems(operationsQueue, workstations, timeOperations, items,
quantMachines);
}
```

- **int quantMachines = workstations.size():** is ***O(1)*** quantMachines is equal to the size of the workstations list. And size() as a complexity of 1.
- **sortMachinesByTime(workstations):** is $\boldsymbol{O(m\log m)}$ because it will get the time of each workstation and compare them between each other, so it goes m times through getTime() and sort() uses an mixture of merge and insertion sort (TimSort), so it will divide the m by 2 until it is able to sort it being why it will go through $\boldsymbol{O(2m\log m)}$ because the getTime() will go through twice to get the value of both workstations so if we simplify it the complexity is $\boldsymbol{O(m\log m)}$.

```java
private static void sortMachinesByTime(ArrayList<Workstation>
workstations) {
    workstations.sort(Comparator.comparingInt(Workstation::getTime));
}
```

- **sortItemsByPriorityAndTime(items, workstations):** is divided into some methods so let's go through them:
- As visualized bellow the joint complexity of **addTimes()** and **swapOperations()** is $\boldsymbol{O(n \cdot k \cdot m + n \cdot k^2 + n\log n)}$.
- So now we need to consider items.sort():
- **items.sort()** will use an comparator that compares all the items in the system based on the list of their lowest times, that were determined before in **addTimes()**, and with a sorted priority that was achieved with **sortItemsByPriority(items)** so we can easily find what are the overall fastest items to complete the process and put them in the list correctly sorted by overall time and there to complete as the list is sorted by time. Now in terms of complexity we have a sort of the items where it is going to sort two different items by the time they take by each of their operations lowest times, and because it uses TimSort (sort method was explained before), so it will have a complexity of $\boldsymbol{O(n\log n)}$ we ignore the fact they do it twice because the complexity stays the same, then it goes into a loop where it goes h times through the loop while also inside comparing the values of both items times

resulting on the use of a compare that uses sort wish leads to the complexity of $\boldsymbol{O(k \log k)}$. So, the sort has an overall complexity $\boldsymbol{O(n \log n \cdot k \log k)}$

- Meaning that the overall complexity of **sortItemsByTime()** is $\boldsymbol{O(n \cdot k \cdot m + n \cdot k^2 + n \log n + n \log n \cdot k \log k)}$.

```java
private static void sortItemsByPriorityAndTime(ArrayList<Item> items,
ArrayList<Workstation> workstations) {
    sortItemsByPriority(items);
    addTimes(items, workstations);
    swapOperations(items);
    items.sort((item1, item2) -> {
        int priorityComparison =
item1.getPriority().compareTo(item2.getPriority());
        if (priorityComparison != 0) {
            return priorityComparison;
        }
        LinkedHashMap<String, Integer> sortedTimes1 =
item1.getLowestTimes().entrySet().stream()
                .sorted(Map.Entry.comparingByValue())
                .collect(LinkedHashMap::new, (map, entry) ->
map.put(entry.getKey(), entry.getValue()), LinkedHashMap::putAll);

        LinkedHashMap<String, Integer> sortedTimes2 =
item2.getLowestTimes().entrySet().stream()
                .sorted(Map.Entry.comparingByValue())
                .collect(LinkedHashMap::new, (map, entry) ->
map.put(entry.getKey(), entry.getValue()), LinkedHashMap::putAll);

        Iterator<Map.Entry<String, Integer>> it1 =
sortedTimes1.entrySet().iterator();
        Iterator<Map.Entry<String, Integer>> it2 =
sortedTimes2.entrySet().iterator();

        while (it1.hasNext() && it2.hasNext()) {
            Map.Entry<String, Integer> entry1 = it1.next();
            Map.Entry<String, Integer> entry2 = it2.next();
            int timeComparison = Integer.compare(entry1.getValue(),
entry2.getValue());
            if (timeComparison != 0) {
                return timeComparison;
            }
        }
        return 0;
    });
}
```

- **sortItemsByPriority()** complexity is $\boldsymbol{O(n \log n)}$ because it will use TimSort, that uses a mixture of merge and insertion sort that is the sort method used by sort() and will also sort all the items by their priority.

```java
private static void sortItemsByPriority(ArrayList<Item> items) {
    items.sort(Comparator.comparing(Item::getPriority));
}
```

- **addTimes(items, workstations):** is ***O(n·k·m)*** assuming n is for items, k for operations and m for workstations as said before. That is going to be the complexity because it will iterate over the items n times then it will iterate over each item operations k times and then it will go through all the machines m times to get the smallest time of each operation that the item has. So there are three loops for each of the objects considered that are inside each other which means that the complexity is O(n·k·m).

```
private static void addTimes(ArrayList<Item> items,
ArrayList<Workstation> workstations) {
    for (Item item : items) {
        LinkedHashMap<String, Integer> operationTimes = new
LinkedHashMap<>();
        for (String operation : item.getOperations()) {
            int minTime = Integer.MAX_VALUE;
            for (Workstation workstation : workstations) {
                if
(workstation.getOperation().equalsIgnoreCase(operation)) {
                    minTime = Math.min(minTime,
workstation.getTime());
                }
            }
            if (minTime == Integer.MAX_VALUE) {
                System.out.println("Warning: No workstation found for
operation: " + operation);
            }
            operationTimes.put(operation, minTime);
        }
        item.setLowestTimes(operationTimes);
    }
}
```

- **swapOperations(items):** is $\boldsymbol{O(n\ \cdot\ k^2)}$ assuming n items and k operations. The program will go through each item n times and then will go through each item operations k^2 times and compare each of those operations until every single operation will be eventually swapped until it is sorted, this could be compared in some way to bubble sort but with a different way of sorting. So the worst case will be **O(n · k^2).**

```
private static void swapOperations(ArrayList<Item> items) {
    for (Item item : items) {
        List<String> operations = item.getOperations();
        LinkedHashMap<String, Integer> lowestTimes =
item.getLowestTimes();
        boolean swapped;
        do {
            swapped = false;
            for (int i = 0; i < operations.size() - 1; i++) {
                String operation1 = operations.get(i);
                String operation2 = operations.get(i + 1);
                if (lowestTimes.get(operation1) >
lowestTimes.get(operation2)) {
                    Collections.swap(operations, i, i + 1);
                    swapped = true;
                }
            }
        } while (swapped);
    }
}
```

Method addAllItems

- Before continuing the method will go through the items n times and create a map with the quantity of each item produced this will be helpful to distinguish items with the same id. The method has a complexity of **O(n).**
- This method will go through all the items n times and entering the addOperations n times. Having the complexity of **O(n) times the complexity of the rest of the program which continues in addOperations().**
- So the complexity is $\boldsymbol{O(n \cdot k \cdot m^2)}$**.** For this method.

```
private static void addAllItems(HashMap<String, LinkedList<Item>>
operationsQueue, ArrayList<Workstation> workstations,
LinkedHashMap<String, Double> timeOperations, ArrayList<Item> items,
int quantMachines) {
    LinkedHashMap<String, Integer> quantItems = new LinkedHashMap<>();
    for (Item item : items) {
        quantItems.put(String.valueOf(item.getId()), 0);
    }
    for (Item item : items) {
        quantItems.put(String.valueOf(item.getId()),
quantItems.get(String.valueOf(item.getId())) + 1);
        if (item.getId() != 0) {
            quantMachines = addOperations(operationsQueue,
workstations, timeOperations, item, quantMachines, quantItems);
        }
    }
}
```

Method addOperations

- This method has a complexity of $\boldsymbol{O(k \cdot m^2)}$ because of the loop that goes through all the operations of a certain item k times and then will go through all the operations and check the ones that have that operation.
- Then it will call **checkMachines()** if the quantity of machines is 0, having a complexity of m as it goes through m times and clear the machines.
- It will also call **checkMachinesWithOperations()** if the quantity is not 0, having a complexity of m as it is explained below.
- And the last method called is **addItem()** that as a complexity of m as it will go through all available items.

```
private static int addOperations(HashMap<String, LinkedList<Item>>
operationsQueue, ArrayList<Workstation> workstations,
LinkedHashMap<String, Double> timeOperations, Item item, int
quantMachines, LinkedHashMap<String, Integer> quantItems) {
    for (String operation : item.getOperations()) {
        ArrayList<Workstation> availableWorkstations = new
ArrayList<>();
        for (Workstation workstation : workstations) {
            if
(workstation.getOperation().equalsIgnoreCase(operation)) {
                availableWorkstations.add(workstation);
            }
        }
        if (quantMachines == 0) {
            quantMachines = checkMachines(workstations,
quantMachines);
        } else {
            quantMachines =
checkMachinesWithOperation(availableWorkstations, quantMachines,
operation);
        }
        quantMachines = addItem(operationsQueue, timeOperations, item,
operation, availableWorkstations, quantMachines, quantItems);
    }
    return quantMachines;
}
```

- The method will check if all the workstations are filled up if yes then it will clear them up this is so there are always machines available in the extreme cases, being why the complexity is just m as it just goes to all the workstations and clears them.

```
private static int checkMachines(ArrayList<Workstation> workstations,
int quantMachines) {

    for (Workstation workstation1 : workstations) {
        workstation1.clearUpWorkstation();
    }
    quantMachines = workstations.size();

    return quantMachines;
}
```

- This method does a similar process to checkMachines but now will just clear up the machines of a certain operation if they are all filled up.
- Because of the similarity it has the same complexity of m, but does two different things being them creating a list of the machines that have that operation and have an item and also seeing if any of those machines still doesn't have an item so it doesn't clear else it will clear.

```
private static int checkMachinesWithOperation(ArrayList<Workstation>
workstations, int quantMachines, String operation) {
    boolean notFree = true;
    int quant = 0;
    ArrayList<Workstation> tempMachines = new ArrayList<>();
    for (Workstation workstation : workstations) {
        if (workstation.getOperation().contains(operation) &&
workstation.getHasItem()) {
            quant++;
            tempMachines.add(workstation);
        }
        if (workstation.getOperation().equalsIgnoreCase(operation) &&
!workstation.getHasItem()) {
            notFree = false;
        }
    }
    if (quant >= 1 && notFree) {
        for (Workstation workstation : tempMachines) {
            workstation.clearUpWorkstation();
        }
        quantMachines = quantMachines + quant;
    }
    return quantMachines;
}
```

- This method has a complexity of m because it goes through all the available machines and adds the item operation.

```
private static int addItem(HashMap<String, LinkedList<Item>>
operationsQueue, LinkedHashMap<String, Double> timeOperations, Item
item, String operation, ArrayList<Workstation> availableWorkstations,
int quantMachines, LinkedHashMap<String, Integer> quantItems) {
    for (Workstation workstation : availableWorkstations) {
        if (item.getCurrentOperationIndex() >
item.getOperations().size()) {
            return quantMachines;
        }
        if
((operationsQueue.get(workstation.getOperation()).contains(item) &&
workstation.getOperation().equalsIgnoreCase(operation)) &&
(!workstation.getHasItem())) {
            int currentItem = timeOperations.size() + 1;
            workstation.setHasItem(true);

item.setCurrentOperationIndex(item.getCurrentOperationIndex() + 1);
            quantMachines--;
            String operation1 = currentItem + " - " + " Operation: " +
operation + " - Machine: " + workstation.getId() + " - Priority: " +
item.getPriority() + " - Item: " + item.getId() + " - Time: " +
workstation.getTime() + " - Quantity: " +
quantItems.get(String.valueOf(item.getId()));
            timeOperations.put(operation1,
timeOperations.getOrDefault(workstation.getOperation(), 0.0) +
workstation.getTime());

operationsQueue.get(workstation.getOperation()).remove(item);
            return quantMachines;
        }
    }
    return quantMachines;
}
```