1. Description of the assignment

The program has to process the sensor data of daily activities. We need to implement and test an application for analyzing the behavior of a person recorded by a set of sensors installed in its house. The historical log of the person’s activity is stored as tuples (start\_time, end\_time, activity\_label), where start\_time and end\_time represent the date and time when each activity has started and ended while the activity label represents the type of activity performed by the person: Leaving, Toileting, Showering, Sleeping, Breakfast, Lunch, Dinner, Snack, Spare\_Time/TV, Grooming.

This kind of information is given in an already defined input file that we will need to read the data from.

It is required that we write a program that uses functional programming in Java with lambda expressions and stream processing to perform the tasks listed (I will further go through these tasks). The results of each task must be written in a separate .txt file (for example, for TASK1 there needs to be a file named task\_1.txt).

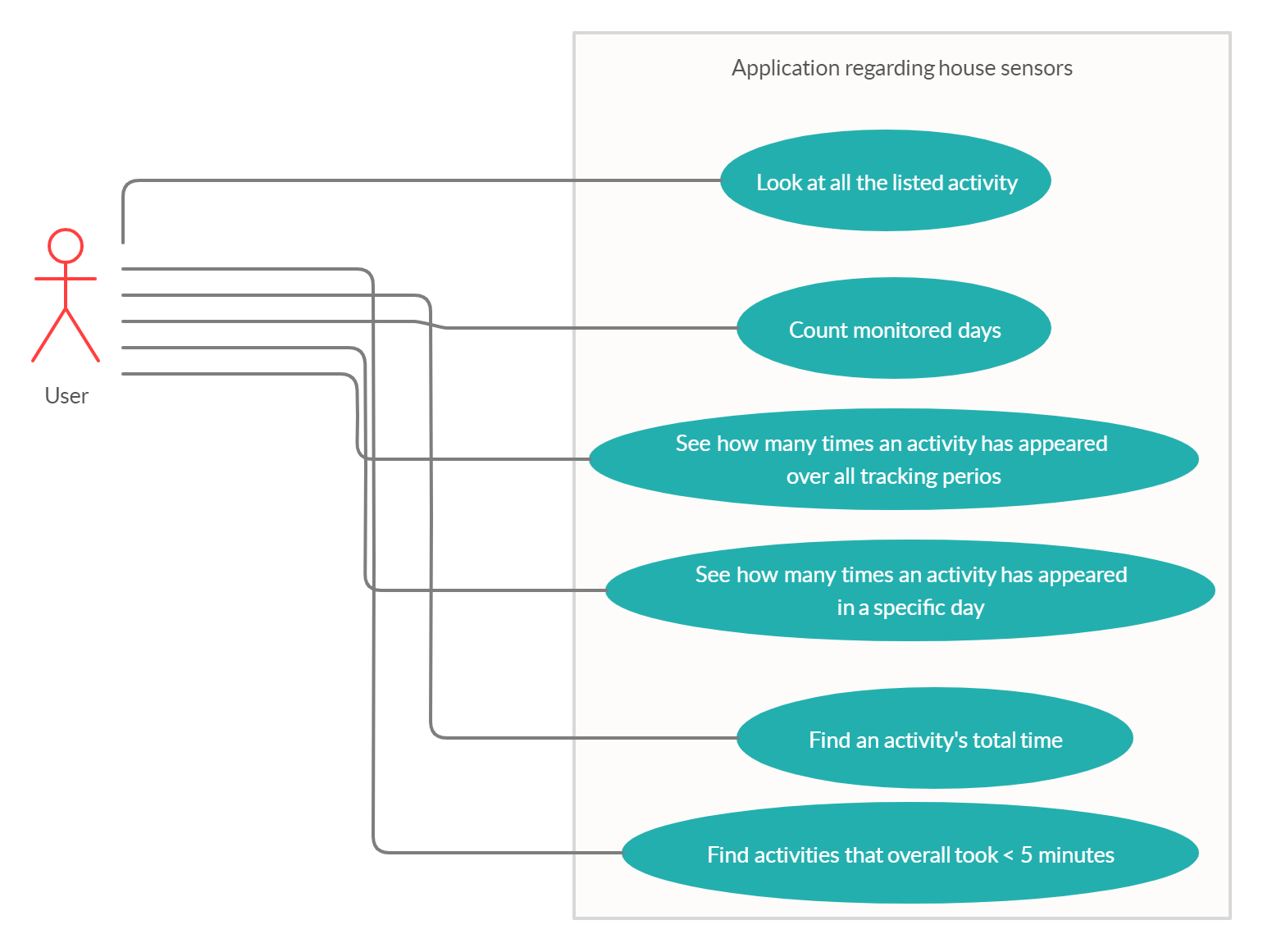
In order to do that, I followed the steps presented in the assignment support file handled to us by our professor and so I did the following (most of them will be presented in detail later into the presentation):

* Defined the class MonitoredData having 3 fields: start time, end time and the activity type.
* Parsed the data from the input file and spilt it into 3 parts so I can create an object of MonitoredData class.
* Added all these new objects in an ArrayList<MonitoredData> and then, for the first task, I made sure I list them into a new .txt file, to be sure I parsed the document as it was requested.
* Counted the distinct days monitored, as required in the second task, and listed their number in a different .txt file.
* Counted how many times one activity has appeared over the entire monitored period, making sure to list each one of them in a new file accompanied by their number of times it appeared.
* Counted how many times one specific activity appears in one specific day and then I listen for each day the duplet (activity, total time of that activity).
* Computed the total duration of an activity over the whole monitoring period and I’ve listed it in a new .txt file (having both the activity and its total duration).
* Filtered the activities that have more than 90% of their tracked time under 5 minutes and I’ve listed these activities along with their period in a new .txt file.
* For a more compressed code, I used the streaming method and lambda expressions
* Provided the .jar file

1. Problem analysis, problem modeling, scenarios, utilization cases

For designing this application, I compressed everything into just one package, named main. Because the program didn’t require any graphical interface or any other input/output mechanism, other than parsing the input file which I incorporated into the class that basically ‘deals’ with the way the application works, I thought the Model View Controller architecture would not be a fitted one, neither the DataLayer-BusinessLayer-PresentationLayer one. That’s the reason why I chose to keep everything in just one package.

To show better how the program is supposed to work, I will present some use-case diagrams next (they will be modeled as lists, showing the steps involved in the execution of each case);



Use Case: going through the data collected by the sensors

Primary Actor: User of the application

Main Success Scenario:

-The user chooses to view one of the six available results

-The user gets the specific .txt document for the action he desires to view(the actions presented in the upper attached diagram)

-The user wants to view another result, so the process repeats itself

Use Case: going through the data collected by the sensors

Primary Actor: User of the application

Alternative Sequences:

-The actions required are not performed well and the .txt files are empty/not created at all

1. Project design (design decisions, UML diagrams, data structures, classes design, relationships, interfaces, packages, algorithms, graphical user interface)

In the following chapter I will discuss how I have split the problem into an object oriented one and the data structures I used for implementing the project, alongside the UML diagrams specific to this application.

Since I’ve already mentioned I have only one package for this application and the reasoning behind my decision, I will go further to attach the UML diagram specific to my project, along with the relationships between them (the following UML diagram is made using the StarUML app);



The only problem with the attached UML diagram is the fact that I couldn’t write Map explicitly, I couldn’t figure out the reason for that, but I decided to at least leave it like that until I can figure out how to properly write its format.

As for the classes and methods used, I will detail them in the following chapter.

1. Implementation

In this chapter I will elaborate on each class I created, explaining the most important methods that each implement and why I chose them the way I did.

* MonitoredData class

This class was required to be created so that it could hold the objects that we are going to create after parsing the input file. Each one of these objects will represent an activity that was monitored, having a type (activity\_label), a starting time as well as an ending time.

The only methods that I needed to implement in this class were toString() which I will later on use for the first task, for listing all the monitored activities (I followed the instruction given to us on the Discord channel and I chose to write the activity as <starting time=… , ending time=… , activity name=… >) and returnStartDate() which will return a string composed of the year-month-day format for each object. The latter method will be used when we count the number of times an activity has appeared in a day (task 4).

* ActivityDuration class

I created this class for it to be a little easier to approach the task where we need to calculate the total duration of an activity, Since the method will return a Map<String, Long>, I thought it would be a good idea to compress the activity in an object of this type, having only the name which will be a String and the duration which will be a Long. Though, the class only has these two parameters (activity\_label and duration) and it will not implement any specific methods since we only need it to store the activities. The way I chose to work with the objects of this class will be presented in the next class explanation.

* PerformTasks class

This is the ‘main’ class of my application, where all the tasks are resolved and the results are written into each specific file.

Firstly, we need to parse through the input file and create the list with the MonitoredData objects. Because two of the parameters of this object are of type LocalDateTime, we will need a formatter to get the right form of the parameter to be added. So, the PerformTasks class will have only one parameter, that being:

**private final static DateTimeFormatter parser = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")**

I chose the .ofPattern method because I wanted to make sure it will take the date correctly, so I gave as parameter exactly the format that our dates are presented in.

Talking about the methods, the first two, processFile(List<MonitoredData> list) and processLine(String line, List<MonitoredData> list), are the ones needed in order to parse the input file and create the list of MonitoredData objects. The first method creates a stream that takes each line of the file and processes it using the second method

**stream.forEach(line -> processLine(line, list))**

where the list parameter represents the list that will be created after the methods are done.

The second method (processLine) splits the string taken according to where it finds \t\t (the distance between two different parameters is of two tabs).

**String splits[] = line.split("\t\t");**

The first will be the starting time, the second will be the ending time and the last one will be the name of the activity. Using these parameters acquired after splitting the array, we will create a new MonitoredData object that will be added to the list.

**LocalDateTime start\_time = LocalDateTime.*parse*(splits[0], *parser*);  
LocalDateTime end\_time = LocalDateTime.*parse*(splits[1], *parser*);  
String activity\_label = splits[2].trim();**

The name of the activity should also be trimmed because there are a few blank space after the string representing the label.

The methods createFile(String name\_of\_file) and writeToFile(String file, String toBeWritten) are the same as in the second assignment and I will use them for each task to create the desired .txt file and to write into it. I will not elaborate these two methods further, considering I’ve already explained them when I first used them (in lab assignment number 2) and their functionality is a simple one also.

Next, for resolving the first task there is the method task1() which will create a new ArrayList<MonitoredData> which will be filled by data from the input file using the parser

**List<MonitoredData> list = new ArrayList<MonitoredData>();  
processFile(list);**

Then, after creating a new file, we go through the list using streaming and write it using the toString method.

**list.stream().forEach(activity ->writeToFile("Task\_1.txt",activity.toString()));**

For counting the distinct days we use the method countDistinctDays (List<MonitoredData> list) which will be implemented using streaming. We map the dates using year-month-day format, creating a string, and we collect them using Collectors (I will further attach the links I used for implementing this idea in the bibliography part of the presentation). Since each mapped string will represent a different date, we can count how many days there are by finding the size of the keyset.

**list.stream().map(data -> data.start\_time.getYear() + "-" + data.start\_time.getMonth() + "-" + data.start\_time.getDayOfMonth()).collect(Collectors.*groupingBy*(Function.*identity*(), Collectors.*counting*())).keySet().size()**

The process for creating and writing the corresponding data into task\_2.txt is similarly to the first one, though I’m not going to discuss it in detail. Only this time, we store the value return from the countDistinctDays() method and we append that value to a string which will be written into the file

**int nr=countDistinctDays(list);  
String s="Total number of counted days: " + nr;  
writeToFile("Task\_2.txt",s);**

For the third task, we map according to the label of the activity, because we need to count how many times and activity has appeared during all the monitoring period. The corresponding value from the map will represent the counted days. Again, we use Collectors to accumulate the data.

**list.stream().map(data -> data.activity\_label).collect(Collectors.*groupingBy*(Function.*identity*(),Collectors.*counting*()))**

To write the results into the file, we will parse through all the strings in the keyset of the map and we will write them (that means we will write each type of activity) accompanied by the value corresponding to each key (that means the total count of the appearances of the activities during the monitoring period).

**for (String activity\_label : countActivityperPeriod(list).keySet())**

**{  
 writeToFile("Task\_3.txt", activity\_label + " has appeared a total of " + countActivityperPeriod(list).get(activity\_label) + " times");  
}**

For counting how many times a specific activity has appeared in a day we will firstly create a new ArrayList of string named DifferentDays. Going through the list of MonitoredData we have created at the first task, we add to the newly created list the string “year-month-day” only if it isn’t already in the list. Doing that will ensure that we have a list containing all different days calculated.

**for(MonitoredData m : list)**

**{  
 if(!DifferentDays.contains(m.start\_time.getYear() + "-" + m.start\_time.getMonth() + "-" + m.start\_time.getDayOfMonth()))  
 DifferentDays.add(m.start\_time.getYear() + "-" + m.start\_time.getMonth() + "-" + m.start\_time.getDayOfMonth());  
}**

After creating the list, we define a new Map<Integer, Map<String, Long>> object named ‘returning’ and we populate it in the following way: going through indices from 1 to countDistinctDays() (which will be the total number of monitored days), and for each string in the list DifferentDays, we attach to the map the key index i (the specific day) and, filtering according to the equality of the current start date (computed using that returnStartDate() I talked earlier about) and the string at position i-1 if the DifferentDays list, we will make the value of the map a Map<String, Long> that contains the activities that were made in that specific day and their count. The thing that grants that the comparison with equals method is always true is the fact that I already know that the dates in the DifferentDays list are arranged from the first one monitored up until the last one. Thus, for the first position we have the day that corresponds to the day 1 and so on.

**for(int i=1; i<=countDistinctDays(list); i++)**

**{  
 int k=i-1;  
 returning.put(i, list.stream().filter(m -> m.returnStartDate().equals(DifferentDays.get(k))).map(data -> data.start\_time.getYear() + "-" + data.start\_time.getMonth() + "-" + data.start\_time.getDayOfMonth() + " - " + data.activity\_label).collect(Collectors.*groupingBy*(Function.*identity*(), Collectors.*counting*())));  
}**

To make sure we map each activity differently, we will consider the string as date + activity name, so that even if the day will be identical for the ones that are in the same day, they will be differentiated by their type.

For writing the result in the file I went through the keyset of the map and used the following:

**for (Integer dayNumber : countActivityperDay(list).keySet())**

**{  
 writeToFile("Task\_4.txt", "For day " + dayNumber + " the activities and their count was: " + countActivityperDay(list).get(dayNumber));  
}**

To resolve the fifth task, I used objects of class ActivityDuration, which I mentioned earlier. First, I created a list of all the converted objects (MonitoredData -> ActivityDuration), which means that the list resulted from the activityDuration() method is similarly to the one resulted after parsing the input file, only this time we don’t need extra data unless it is the name of the activity and its duration in minutes.

**return list.stream().map(data -> new ActivityDuration(data.activity\_label, durationInMinutes(data.start\_time, data.end\_time))).collect(Collectors.*toList*())**

The method encountered, durationInMinutes() is described as the converted from LocalDataTime to Long using ChronoUnit. I will attach the links that helped figure this out in the bibliography part of the presentation.

**public Long durationInMinutes(LocalDateTime start, LocalDateTime end)**

**{  
 return start.until(end, ChronoUnit.*MINUTES*);  
}**

After we’re sure that we have our desired list, with name and duration of activity expressed in minutes, the method totalDuration() executes the desired task. Going through the list with a stream, we group everything by their activity\_label and we sum the duration, in such a way that we will get the total duration of the activity.

**activityDuration(list).stream().collect(Collectors.*groupingBy*(data -> data.activity\_label, Collectors.*summingLong*(data -> data.duration)))**

The output file for this task will be created in the following way (no further explanations are needed because it resembles the ones that I already presented):

**for (String activity : totalDuration(list).keySet())**

**{  
 writeToFile("Task\_5.txt", "For the activity " + activity + " the total duration was " + totalDuration(list).get(activity) + " minutes");  
}**

For the last task, I will use the one I just presented. First, I will create a new Map<String, Long> which will be initialized with the one we just computed, totalDuration(). That means we have each activity and its total duration during the monitoring period. Creating an empty ArrayList<String> shortActivities, we will add just the activities that have 90% of the total time smaller than 5 minutes in the following way: for each activity from the Map we have, we calculate if 0.9\*its total duration is smaller than 5 minutes. If so, add it to the list of short activities.

**for(String activity: totalduration.keySet())**

**{  
 if(0.9 \* totalduration.get(activity) <= 5)**

**{  
 shortActivities.add(activity);  
 }  
}**

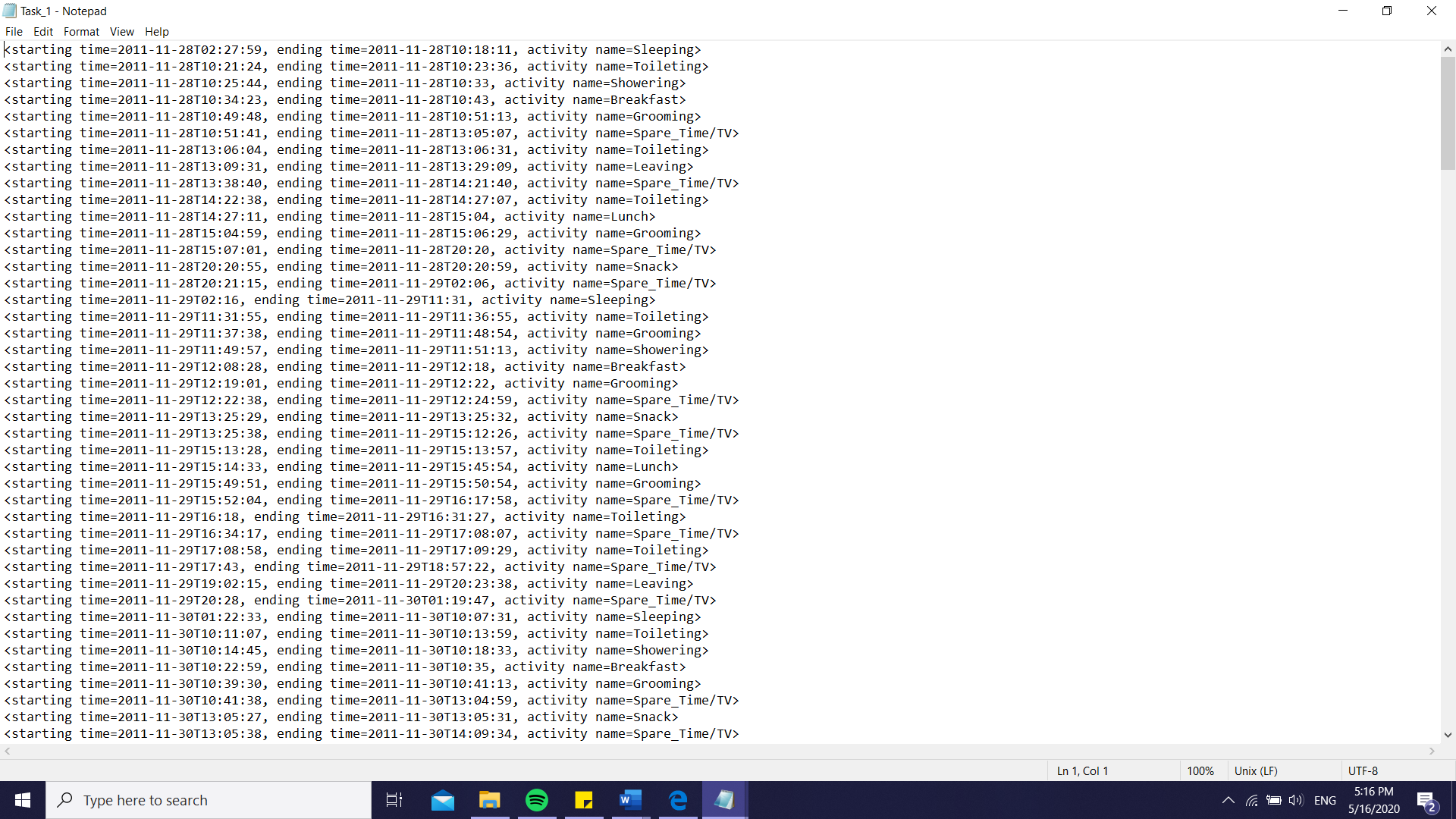
To write the output, I will use a similar method as the ones already presented:

**for (String activity : ActivitiesLessThan5(list))**

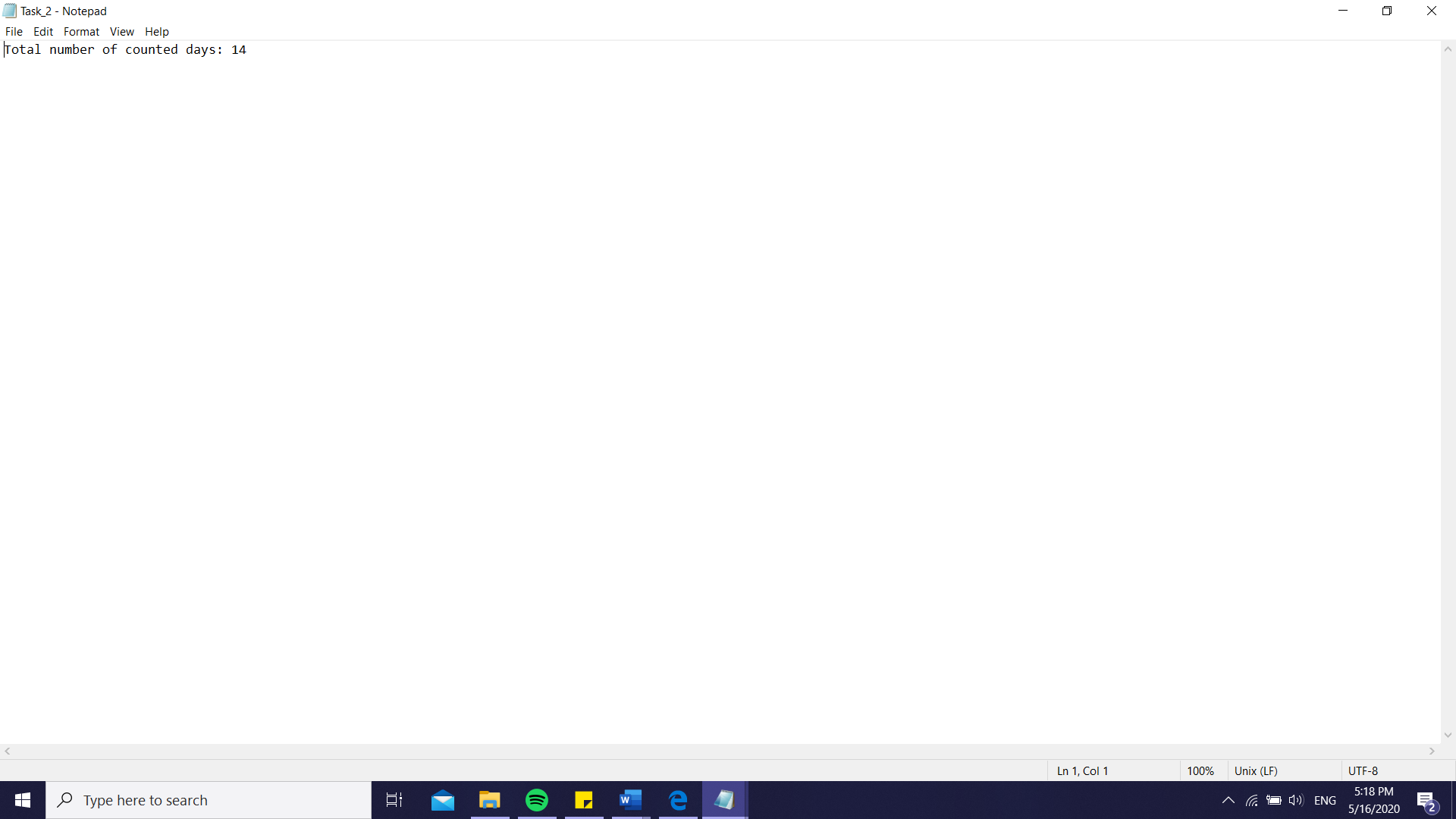
**{  
 writeToFile("Task\_6.txt", "For the activity " + activity + " the total duration was " + totalDuration(list).get(activity) + " minutes, which means 90% of the monitoring records with duration less than 5 minutes");  
}**

1. Results

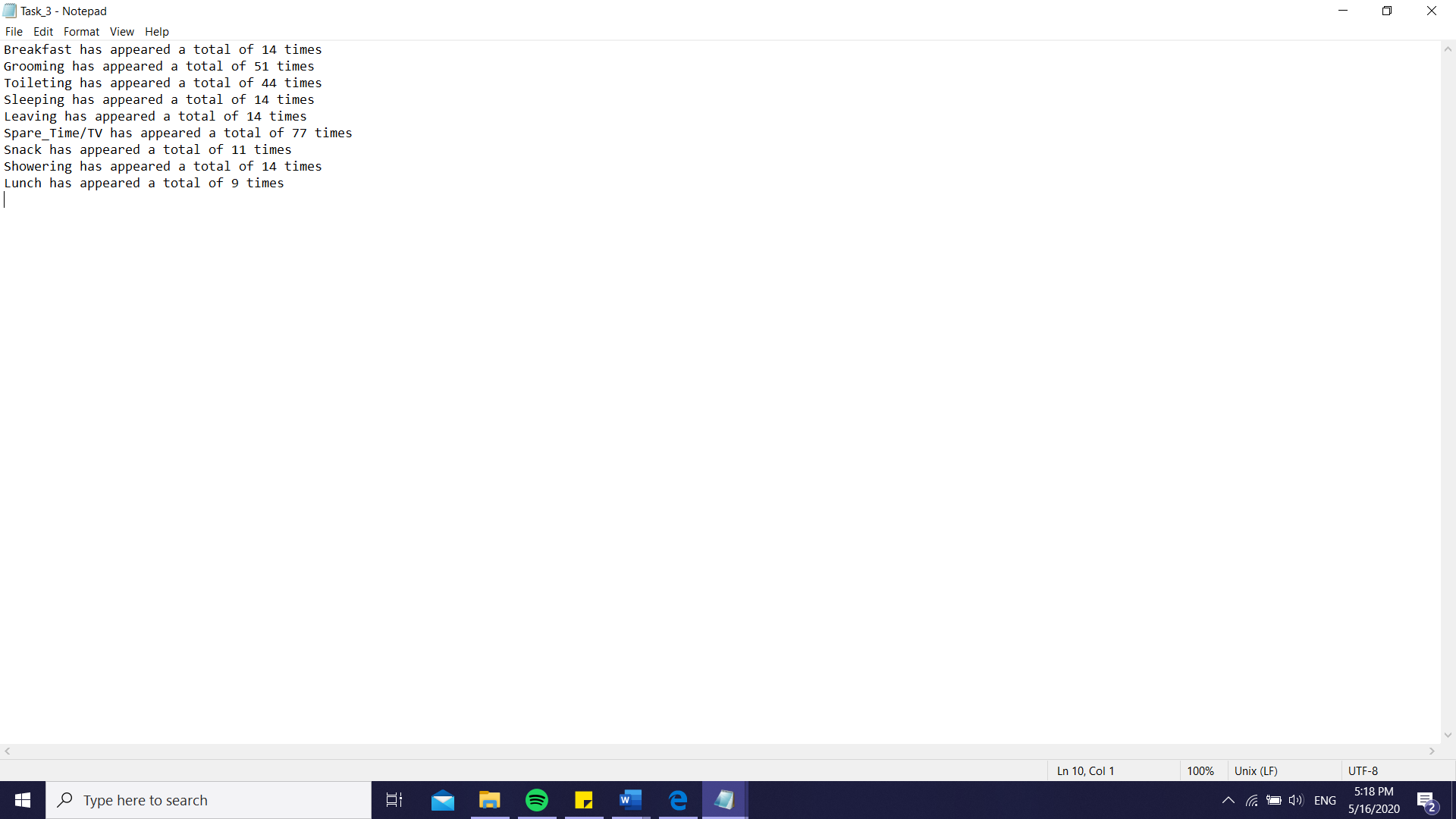
I will further attach some screenshot taken during the running time of the application.



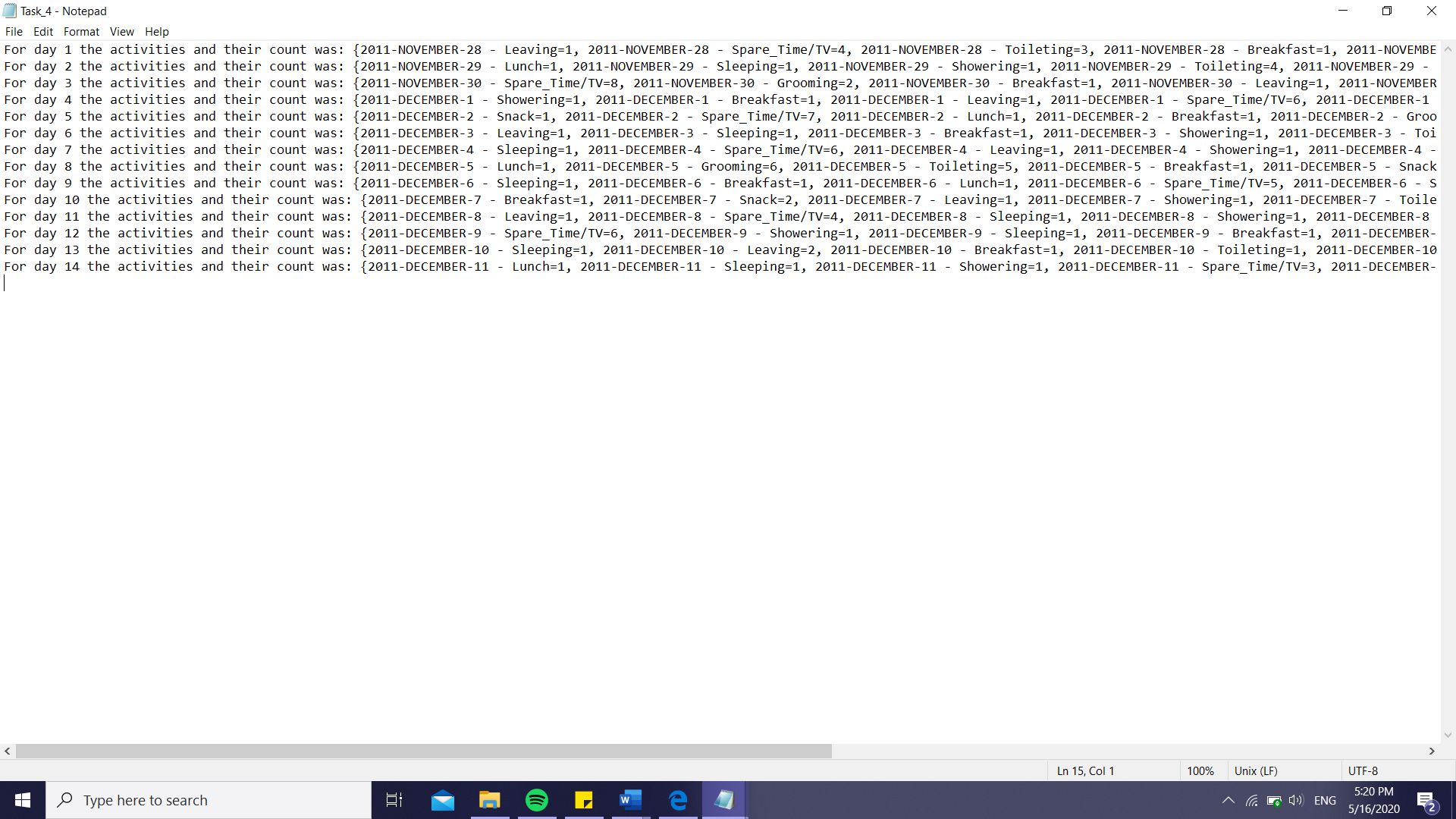
The first output file contains all the activities with their starting date and ending date, in the format I earlier presented.



The second output file contains the total number of days for monitoring.

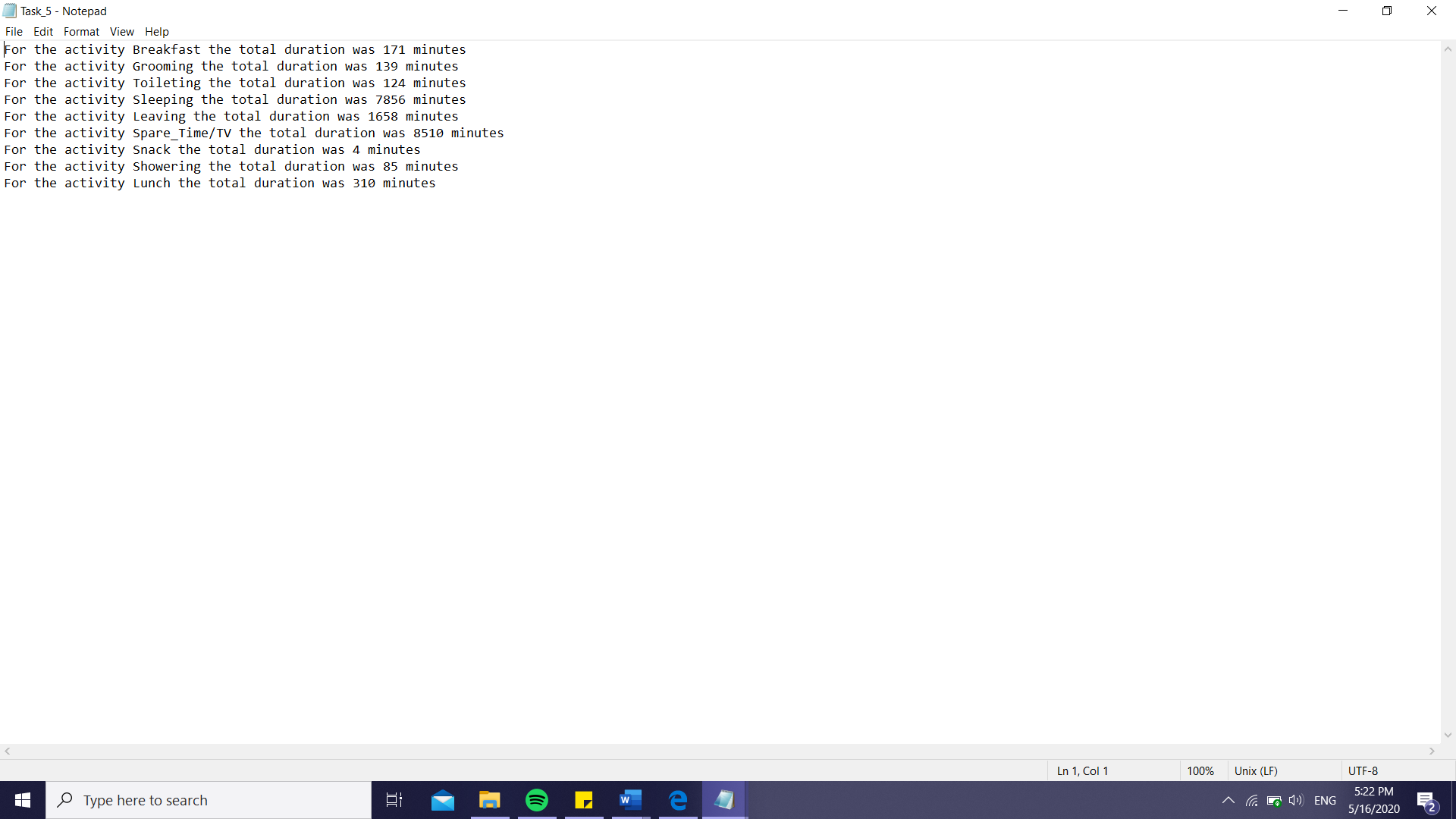


The third output file contains the total times an activity appears during the monitoring period.

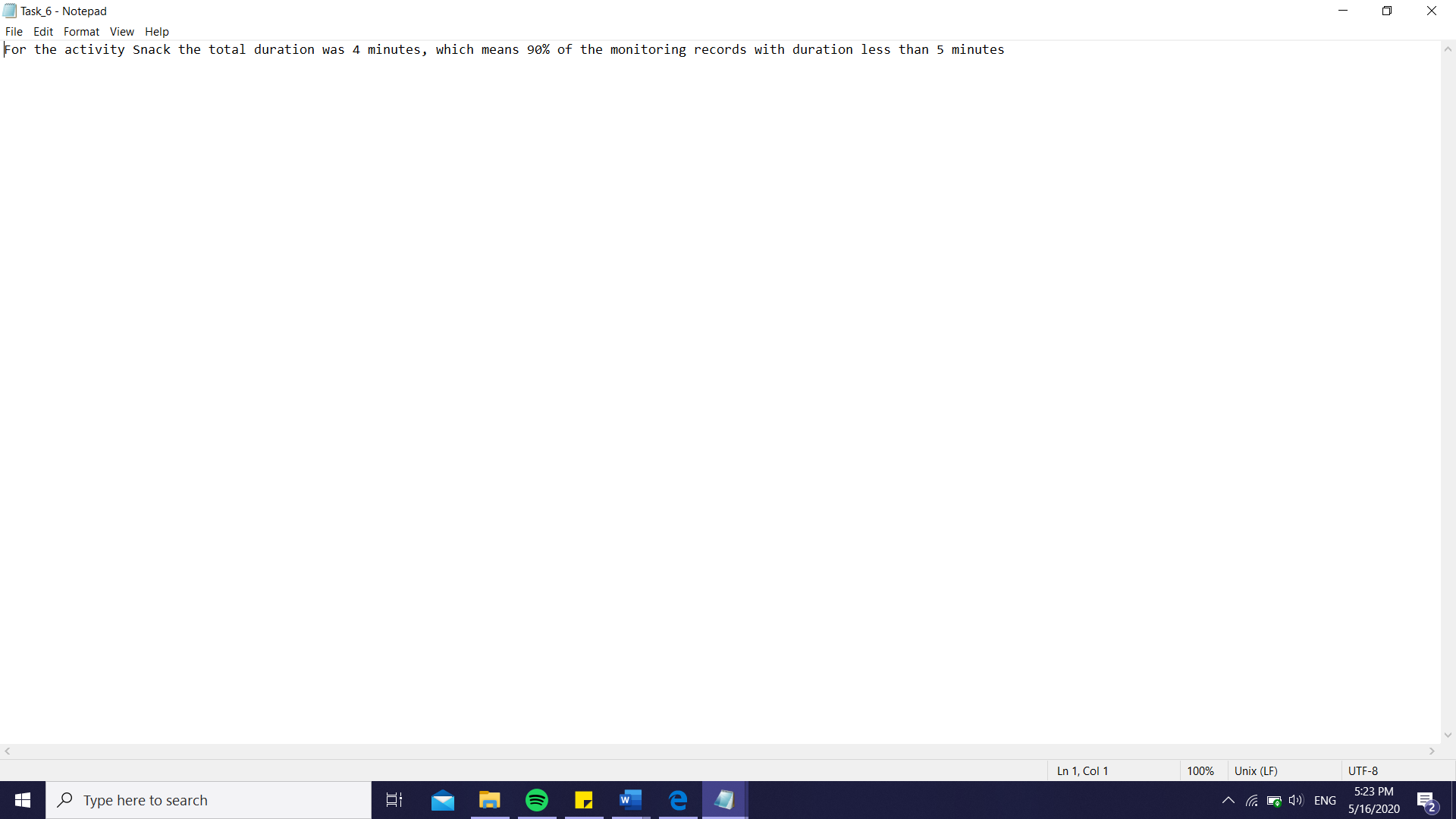


The fourth output shows each day and the activity count specific to a day.

I could further work on the format of the output (The date appears each time for a new activity label, I think it could be improved in such a way to be shown only once).



The fifth output presents the total duration in minutes for each activity.



The sixth output present the activities that took overall less than 5 minutes (90% of the time).

1. Conclusions

In conclusion, I found this assignment the easiest so far, because of the easy way to work with lambda expressions. Even if they were something new to me, I didn’t find them hard to understand or to use. The code was not that complex either, the logic behind the tasks being a simpler one compared to the other assignments, in my opinion. Also, because of the compressed way in which streaming and lambda expression work, I think this newly acquired knowledge will help me a lot for my future project, giving me a better coding style.

1. Bibliography

I will attach some links I felt were useful for my work within this project, as well as some mentions I thought I should make:

* Firstly, I though the course presented by Mr. Salomie on lambda expressions was really helpful, even though I don’t any link to attach regarding it
* In addition to that, I used the following links to learn more about lambda expressions

<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>

<http://tutorials.jenkov.com/java/lambda-expressions.html#single-method-interface>

* To better understand method references, I used the information found here:

<https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html>

* To take a handle of how it will be working with streams, I used the following information

<https://www.oracle.com/technical-resources/articles/java/ma14-java-se-8-streams.html>

<https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/>

* To read a little bit about LocalDataTime variable, I used the link:

<https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html>

* For parsing the LocalDataTime objects I used the helpful links:

<https://stackoverflow.com/questions/22463062/how-to-parse-format-dates-with-localdatetime-java-8>

<https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html>

* For the mapping I used when I counted the distinct days and further when counting how many times an activity appears in a specific day, I used the guide below:

<https://stackoverflow.com/questions/25441088/group-by-counting-in-java-8-stream-api>

* For getting acquainted with the concept of ChronoUnit, I used the link:

<https://docs.oracle.com/javase/8/docs/api/java/time/temporal/ChronoUnit.html>