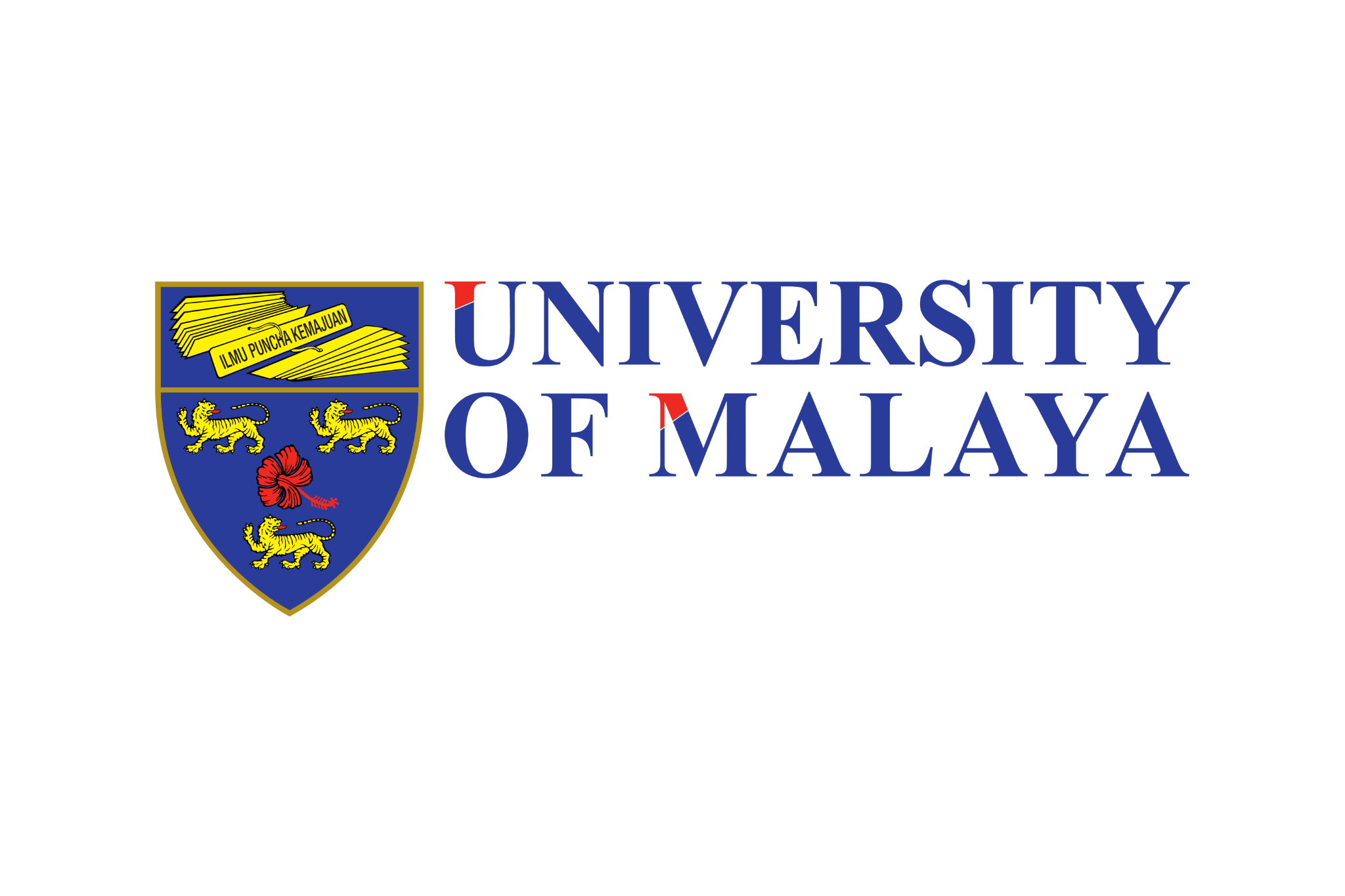
****

**WIX 1002 - Fundamentals of Programming**

**Semester 1**

**Session 2022/2023**

**Lab Assignment Report**

**Instructor Name: Dr. Liew Chee Sun**

**Prepared by:**

**Group 6:**

| **NAME** | **MATRIC NO.** |
| --- | --- |
| **TIEW CHEE YAN** | **22057371** |
| **ERIC LEE CHUN KIAT** | **22004777** |
| **TAN YI SHAN** | **22004867** |
| **YAP WEI LI** | **22004749** |
| **ADAM TAN YONG XUAN** | **22004889** |

**CONTENTS:**

| **NO.** | **DESCRIPTION** | **PAGE** |
| --- | --- | --- |
| **1.0** | **Introduction** | **2** |
| **2.0** | **The formation of team** | **3** |
| **3.0** | **Role and Assigned work for each of the members** | **4** |
| **4.0** | **Outcome and Findings** | **5 - 66** |
| **5.0** | **Problems & Solutions** | **67** |
| **6.0** | **Other problems arise** | **68** |
| **7.0** | **Extra Features created** | **69 - 71** |
| **8.0** | **Conclusion** | **72** |

**1.0 Introduction**

HPC which is called High Performance Computing is a practice of harnessing the power of a supercomputer to solve the problems which involves massive calculation and a huge volume of data. It can be done by connecting with several servers and controlled by a set of cluster management middleware and supporting systems. University of Malaya also has used HPC to solve the computing problem and users including UM researchers and students can access UMHPC by creating a DICC account from the official DICC website.

In this assignment, a log file which contains the HPC information from June to December has been assigned to every team. Every team is considered to extract useful data from the file which can be further used for computing problems. The data extracted can help the users to find out the problems and the information they want.So, too tackle this assignment, we decided to write a program in java language to extract all the information and data needed. The program will be able to read the log files and is equipped with a simple user interface to ease the usage of the user. Additionally, the program can also display simple graphs and tables to visualize the data extracted so that users can analyze it easier and faster.

**2.0 The formation of team**



**3.0 Role and Assigned work for each of the members**

| **MEMBERS** | **ASSIGNED WORKS** |
| --- | --- |
| **YAP WEI LI**   * Leader * Lead-Programmer * Lead-Product Tester * Report editor. | * Assigned the tasks to group members. * Compile all codes from group members. * Manage the project and set a deadline. * Ensure all group members execute their programs well. * In charge of creating Graphical User Interface (GUI) and UI. * In charge of data visualization and creating graphs or charts. * Coding for errors caused by users. * Writing report. |
| **TIEW CHEE YAN**   * Sub-Programmer * Sub-Product Tester * Report editor | * Coding for extra features, kills job. * Coding for number of jobs creating errors and corresponding users. * In charge of data visualization and creating graphs or charts. * Writing report. |
| **Eric Lee Chun Kiat**   * Sub-Programmer * Sub-Product Tester * Report editor | * Coding for extra features,kill job. * Coding for jobs created and ended. * In charge of data visualization and creating graphs of charts. * Writing report. |
| **TAN YI SHAN**   * Sub-Programmer * Sub-Product Tester * Report editor | * Coding for the number of jobs by partitions. * Coding for jobs created and ended. * In charge of data visualization and creating graphs or charts. * Writing report. |
| **ADAM TAN YONG XUAN**   * Sub-Programmer * Sub-Product Tester * Report editor | * Coding for average execution time of jobs submitted to UMHPC. * In charge of data visualization and creating graphs or charts. * Writing report. |

**4.0 Our outcomes and findings**

**SOURCE CODE LINK:**

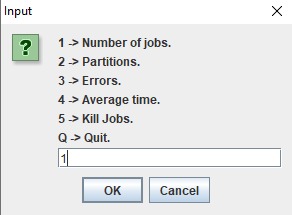
[**https://github.com/Faker000000000/FOP-ASSIGNMENT**](https://github.com/Faker000000000/FOP-ASSIGNMENT)

**(A) Number of jobs created/ended within a given time range**

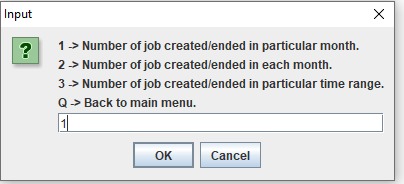
**Source Code (Input link):**

[Source code of number of jobs created/ended](https://docs.google.com/document/d/1-vAVwqEYq23ZkCl0fyJQMvtJsIyhFb2vQacb1XmHe_4/edit?usp=sharing)

**Sample Output:**



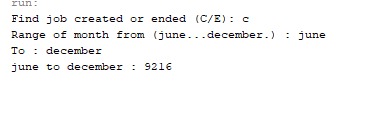
*Enter ‘1’ in the blank to enter the number of jobs created/ended*



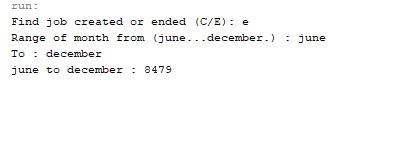
*A window with a list of commands will pop out. The window has three options representing three ways of output. Users just need to enter ‘1’/‘2’/’3’ for different outputs.*

1. Number of job created/ended in particular month

After entering ‘1’, the system will ask users looking for the number of jobs created/ended. After that, users need to enter the range of month they want and the output will show the total number of jobs created/ended within the month range.

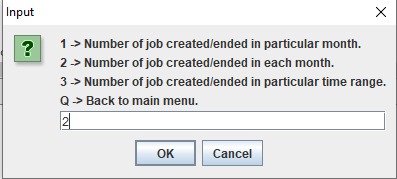


*Enter ‘c’ or ‘C’ is for jobs created. For the picture above, the output will be the total number of jobs created from June to December.*

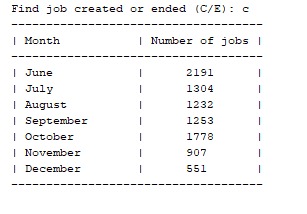


Enter ‘e’ or ‘E’ is for jobs ended. For the picture above, the output will be the total number of jobs created from June to December.

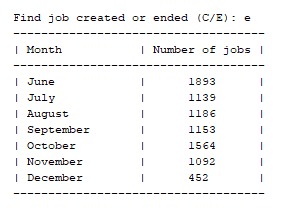
1. Number of jobs created/ended in each month



*Option ‘2’ is for the number of jobs created/ended in each month.*

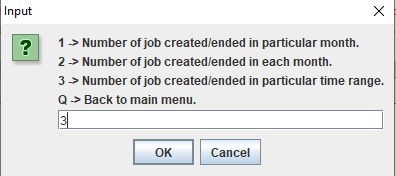


*This table show the number of jobs created for each month after enter ‘c’ or ‘C’*

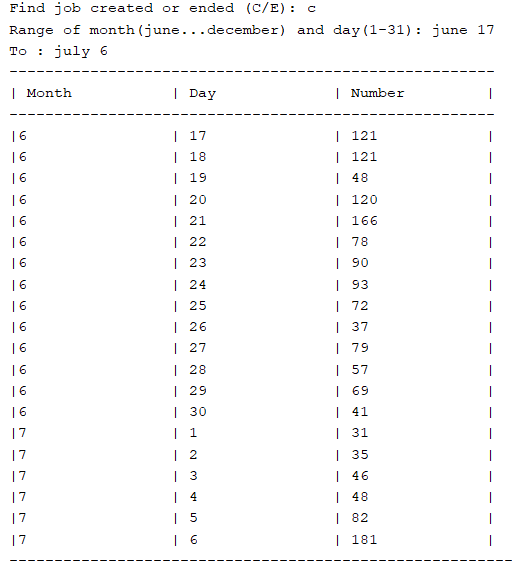


This table show the number of jobs ended for each month after enter ‘e’ or ‘E’

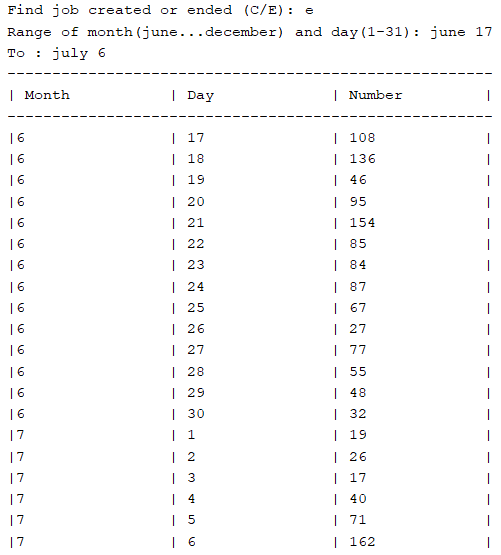
1. Number of jobs created/ended in particular time range



*Option ‘3’ is for the number of jobs created/ended in a particular time range.*

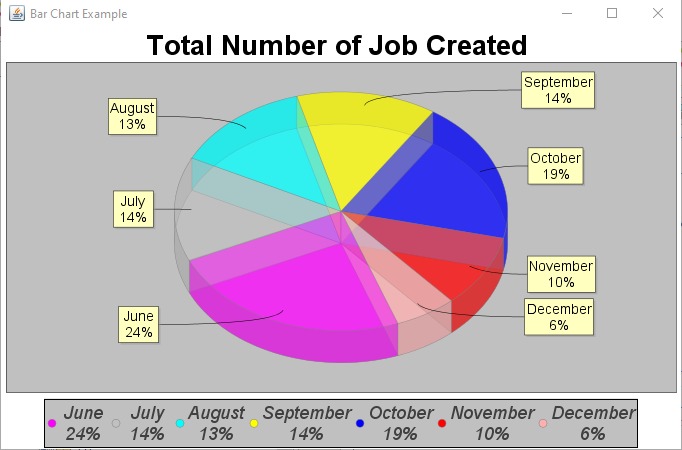


*This table shows the number of job created each day from 17 june to 6 july*

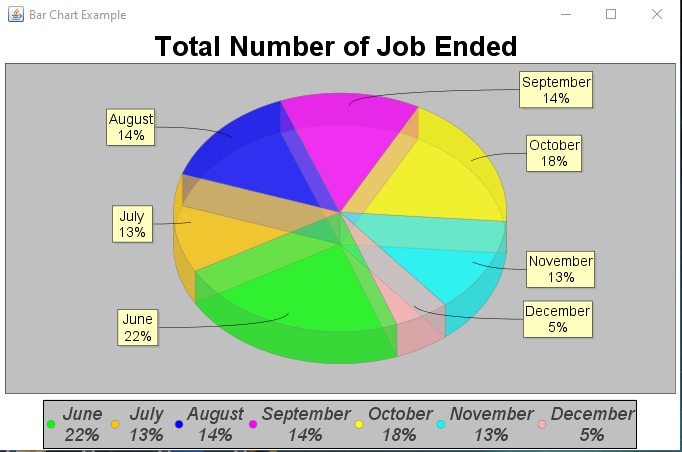


*This table shows the number of job ended each day from 17 june to 6 july*

**Graph/Chart:**



*Display a pie chart about the total number of jobs created*



*Display a pie chart about the total number of Jobs Ended*

**Description:**

To extract the number of jobs created/ended in a particular month, **jobRange()** method has been used. Firstly, the program will ask the users looking for jobs created or ended to identify the pattern which is needed to check whether the line in the file is the line for job created or ended. After that, the program will ask users to enter the month range users are looking for. Then, the program will read the raw data from the text file using **Scanner**. If the line is the jobs created or ended and the line is between the range of month. The variable count will increase by one. If the date of the line in the text file is after the maximum of month range, the loop will break. Finally, the output of the total number of jobs created or ended between the month range will be printed.

To extract the number of jobs created/ended in each month, **allJobs()** method has been used. Firstly, the program will ask the users looking for jobs created or ended to identify the pattern which is needed to check whether the line in the file is the line for job created or ended. Then, the program will read the raw data from the text file using **BufferedReader**. After that, the program will match whether the line read is jobs created or ended. If yes, the number inside the **countAmount** array will increase by one according to the month. Finally, the output of each month will be shown in the table and the program calls the method to create a pie chart .

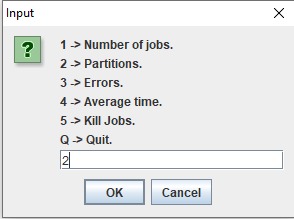
To extract the number of jobs created/ended in a particular time range, **JobRangeDay()** method has been used. Firstly, the program will ask the users looking for jobs created or ended to identify the pattern. After that, the program will ask users to enter the time range users are looking for. Then, the program will read the raw data from the text file.After that, the program will match whether the line read is jobs created or ended. If yes, the program will count the total amount of the particular line on the same date and put the amount into the variable named **count**. The program uses **if-else** statements to check the date range, compare dates, and handle edge cases such as the last day of the month. If the date of the line in the text file is after the maximum of the final date, the loop will break. Finally, the total number of jobs created or ended in the range of a particular date will be printed in the form of a table. The table shows the month,day and number of jobs created/ended.

**(B) Number of jobs by partitions**

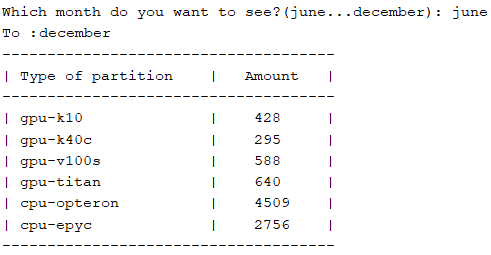
**Source Code (Input link):**

[Source code of partition](https://docs.google.com/document/d/1z5MSIF9nTp9Eo6zQjqw5YChqWuf0JuN7mRyM0btehyQ/edit?usp=sharing)

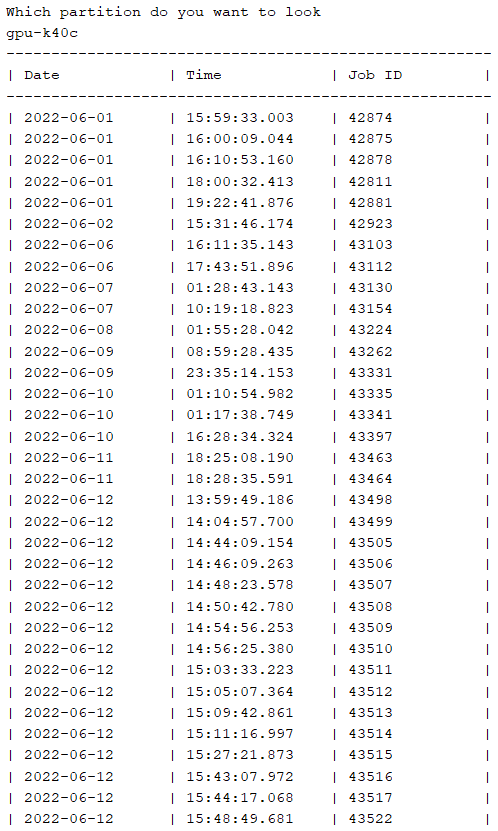
**Sample Output:**

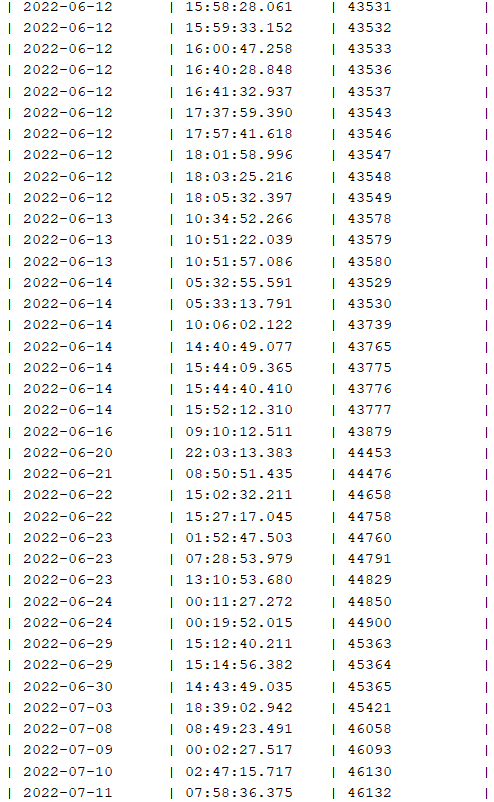


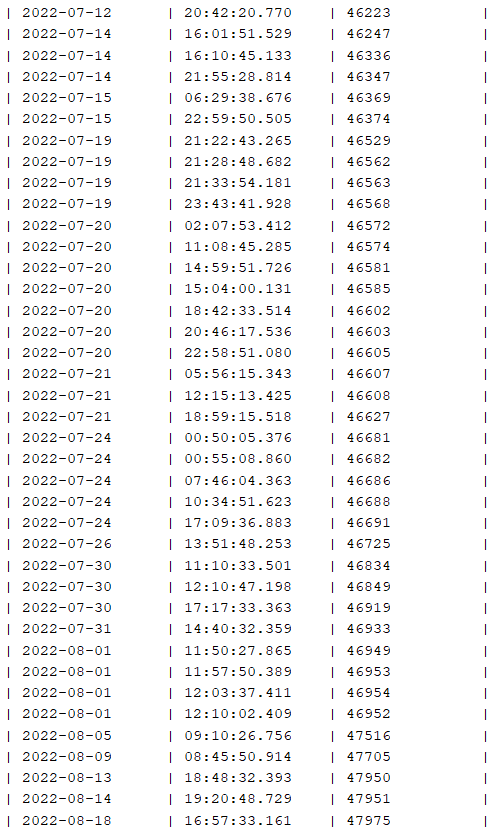
*Enter ‘2’ in the blank to enter the number of jobs by partitions*

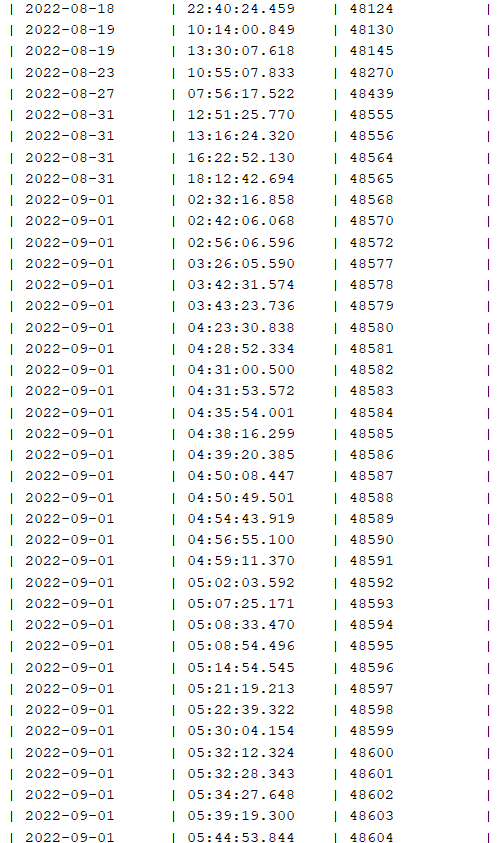


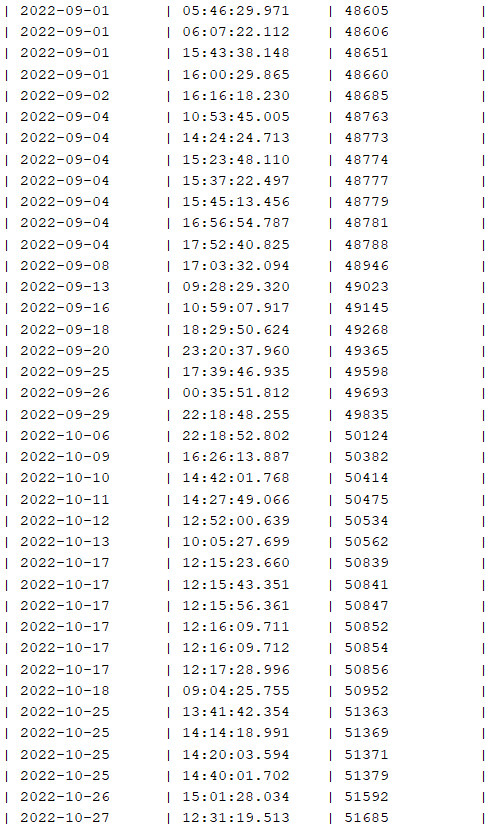
*The table above shows the total number of each partition from June to December after the users enter the month range.*

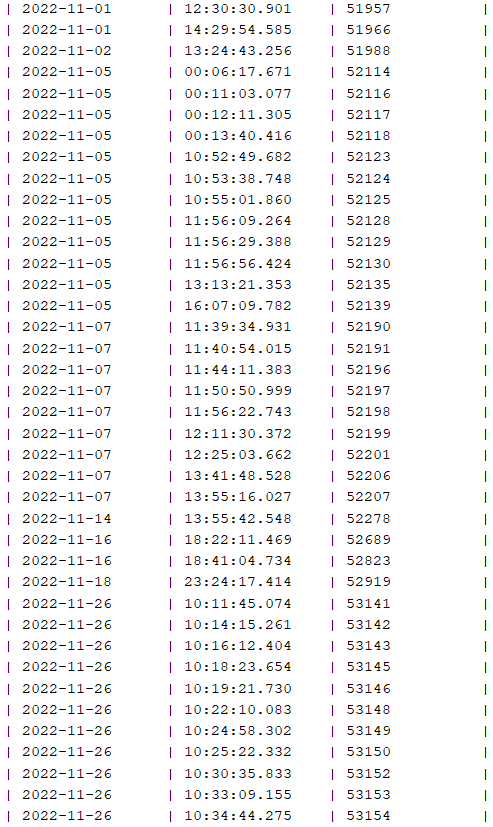


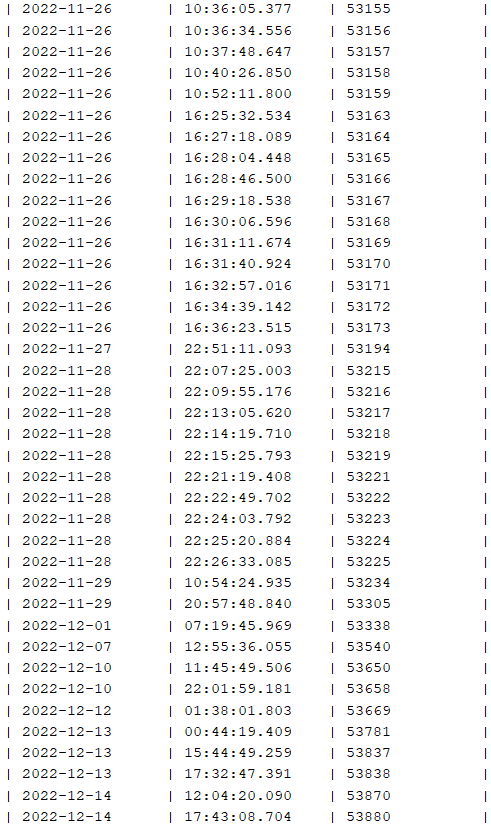


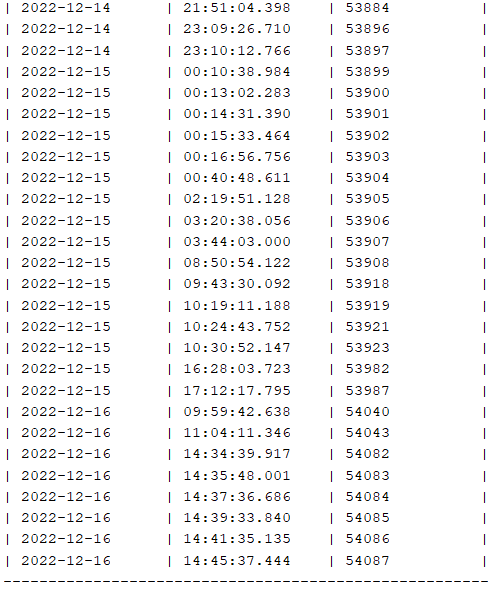






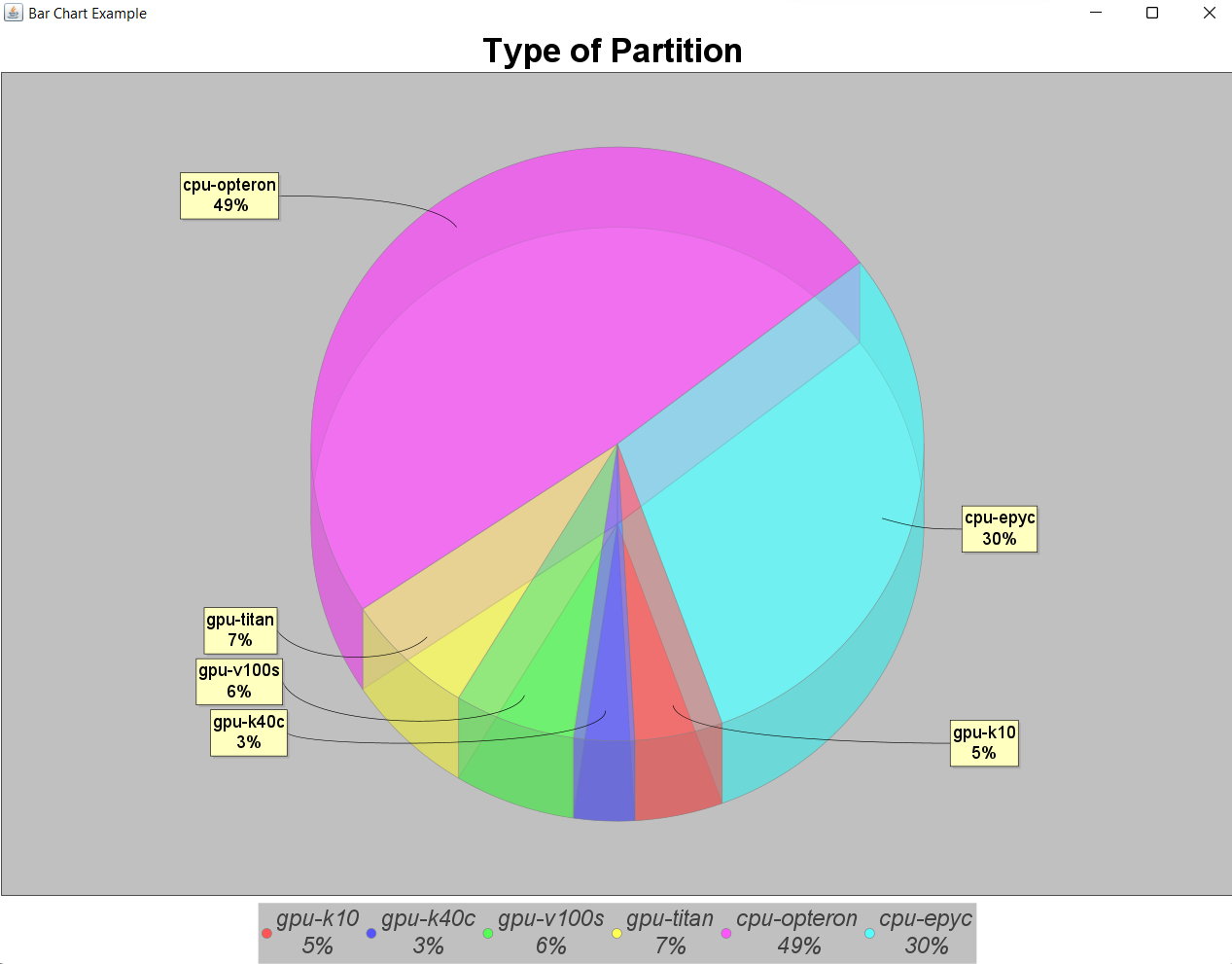






*The table above shows the date, time and job ID of the particular partition after the user enters the partitions.*

**Graph/Chart:**



*Display a pie chart about the number of each partition from June to December*

**Description:**

To extract the number of jobs by partitions, **partition()** method has been used. Firstly, the program will ask users to enter the month range users are looking for. Users can decide the program extract the data from one month to another month. Then, the program will read the raw data from the text file using **Scanner**. After that, the program will check whether the line is between the time range or not. If within the time range, it will match the line whether the line is the code needed by using the regular expression to check. If still yes, the program will increase the number by 1 according to the type of partitions by use for loop. The number of each partition will be kept in the **numPartition** array. If the date of the line in the text file is after the maximum of month range, the loop will break. Finally, the output of each partition will be shown in the table and the program calls the method to create a pie chart if the range is from June to December.

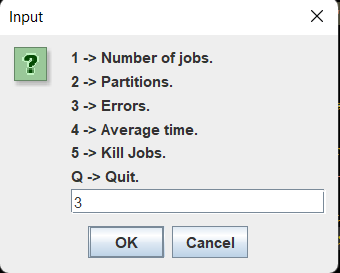
After that, users can enter the types of partitions they are looking for to have more details and **partitionDetail()** method will be called. The method will read the partition users are looking for, and the month range from **partition()** method as the parameters. Then, the program will read the raw data from the text file using Scanner. After that, the program will check whether the line is between the month range or not. If within the month range, it will match the line whether the line is the code needed by using the regular expression to check. If the line is within the month range and the partition written in line is the same as the users are looking for, the date, time and Job ID will be displayed in the form of a table by using regular expression to extract.

***(C) Number of job causing error and the corresponding user***

**Source Code (Input link):**

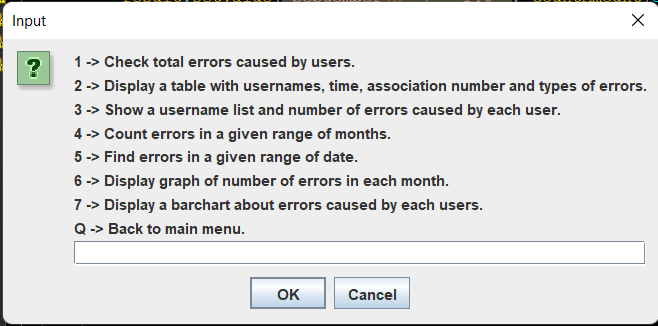
[Source code for error part](https://docs.google.com/document/d/1-UxZhfEZ4sfFCeltCSfvpo-KS3u7TattmSHVzRyYcnw/edit?usp=sharing)

**Sample Output:**



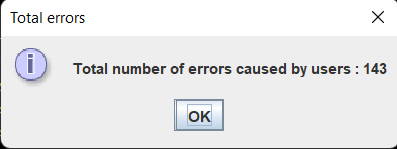
Choose 3 at the main menu to enter the errors part.

After choosing 3 :



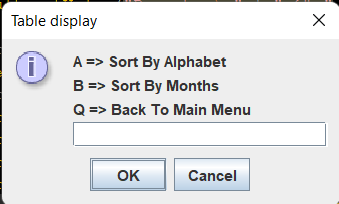
*A window with a list of commands will pop out. There are seven commands we can input, each with different outputs as shown in the picture above.*

***1)Check total number of errors caused by users:***

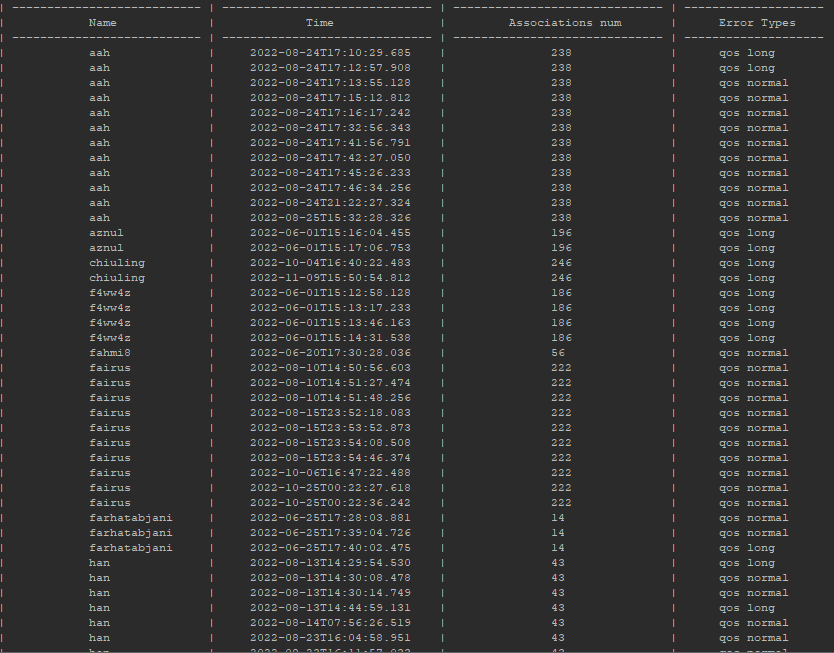


*\*A window will pop out and show the total number of errors caused by users.*

***2) Display table:***

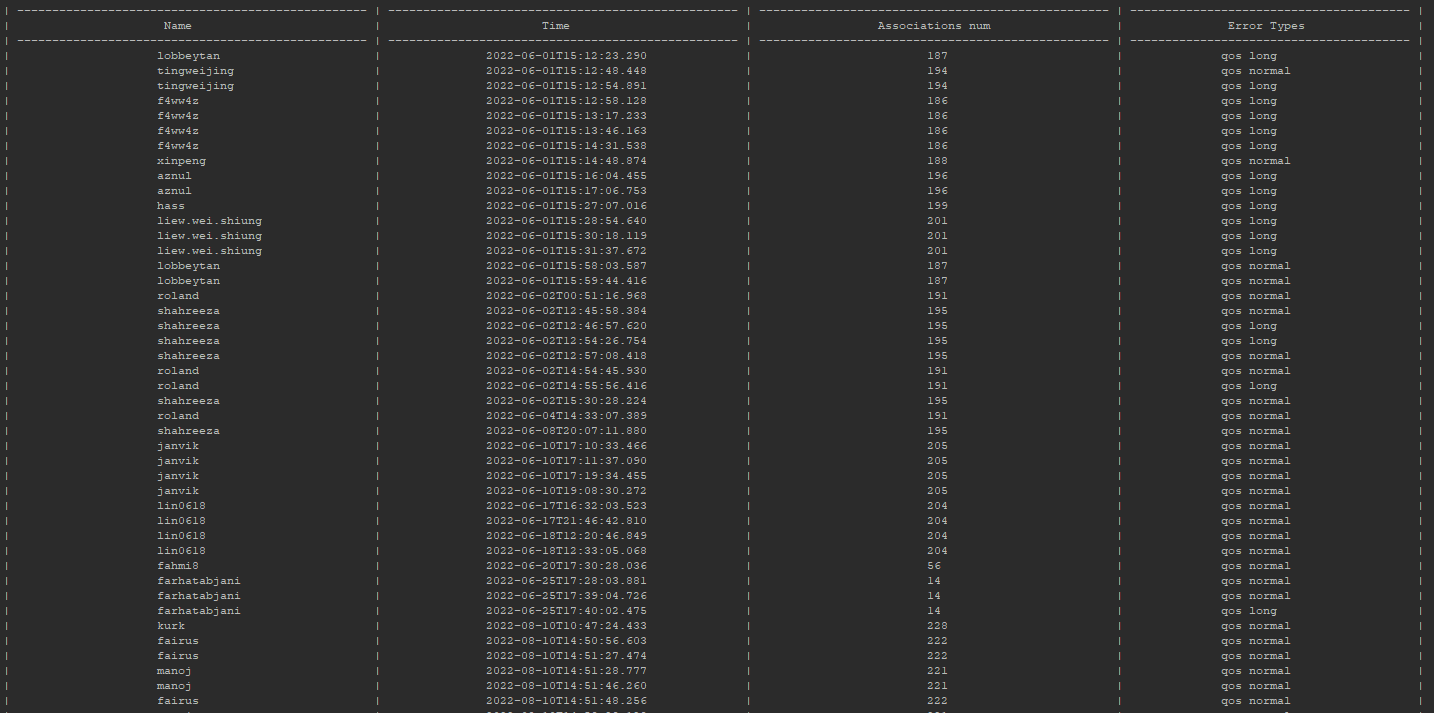


There are two options after we input command 2, which is to display a table with information sorted by Alphabet and sorted by months/time :

A) Sort by Alphabet :

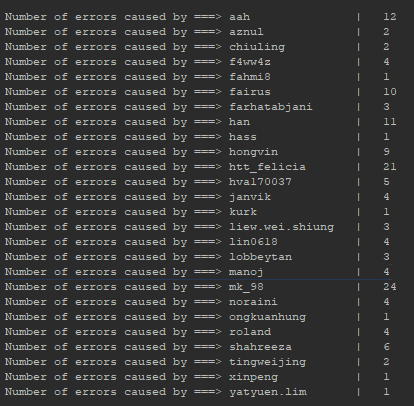
*\*This is a table which prioritizes the sorting of usernames by alphabet, which means the information is displayed according to the alphabet of usernames.*

B) Sort by Months :



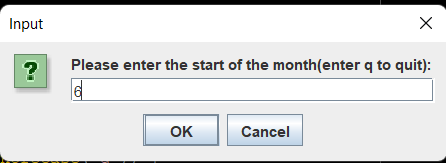
*\*This is a table which prioritizes the sorting of time, which means information is displayed according to time.*

***3) Number of errors caused by each user :***

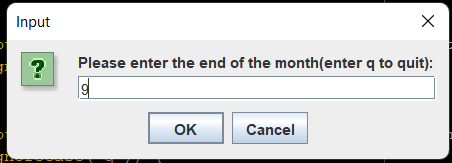


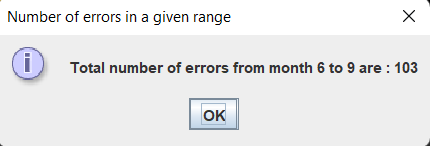
***4) Number of errors within two months:***

-*The user will be prompted to input the start of the month as below:*



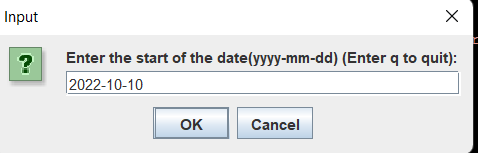
-*Then, user will be prompted to input the end of the month:*





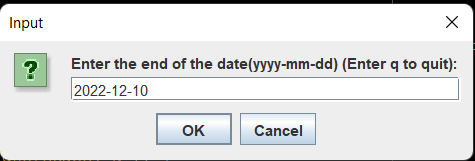
*\*As a result, a window will pop out to show the total number of errors within the range.*

***5) Number of errors within a start and an end date:***

-*User will be prompted to input the start date in correct format:*

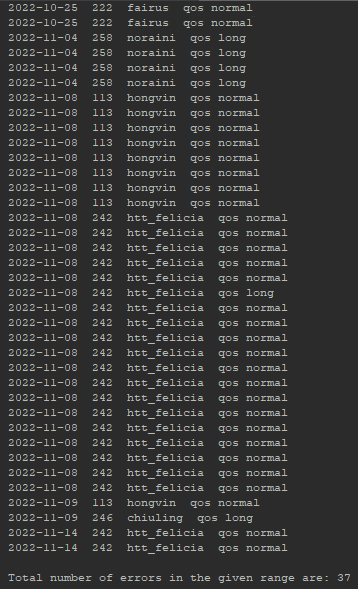
*\*Here we input 2022-10-10 as the start date.*

-*Then, user will be prompted to input the end of the date in correct format:*



*\*Here we input 2022-12-10 as the end date.*

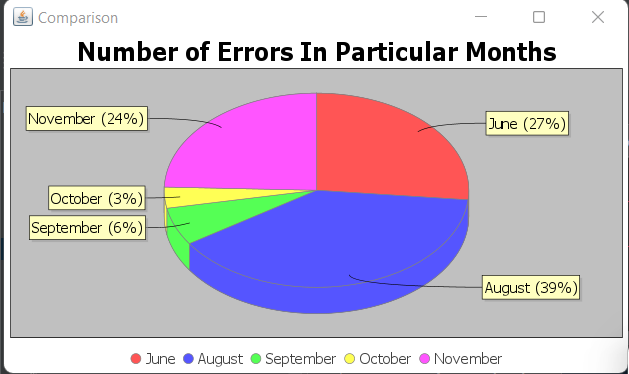
The result is as shown as below:



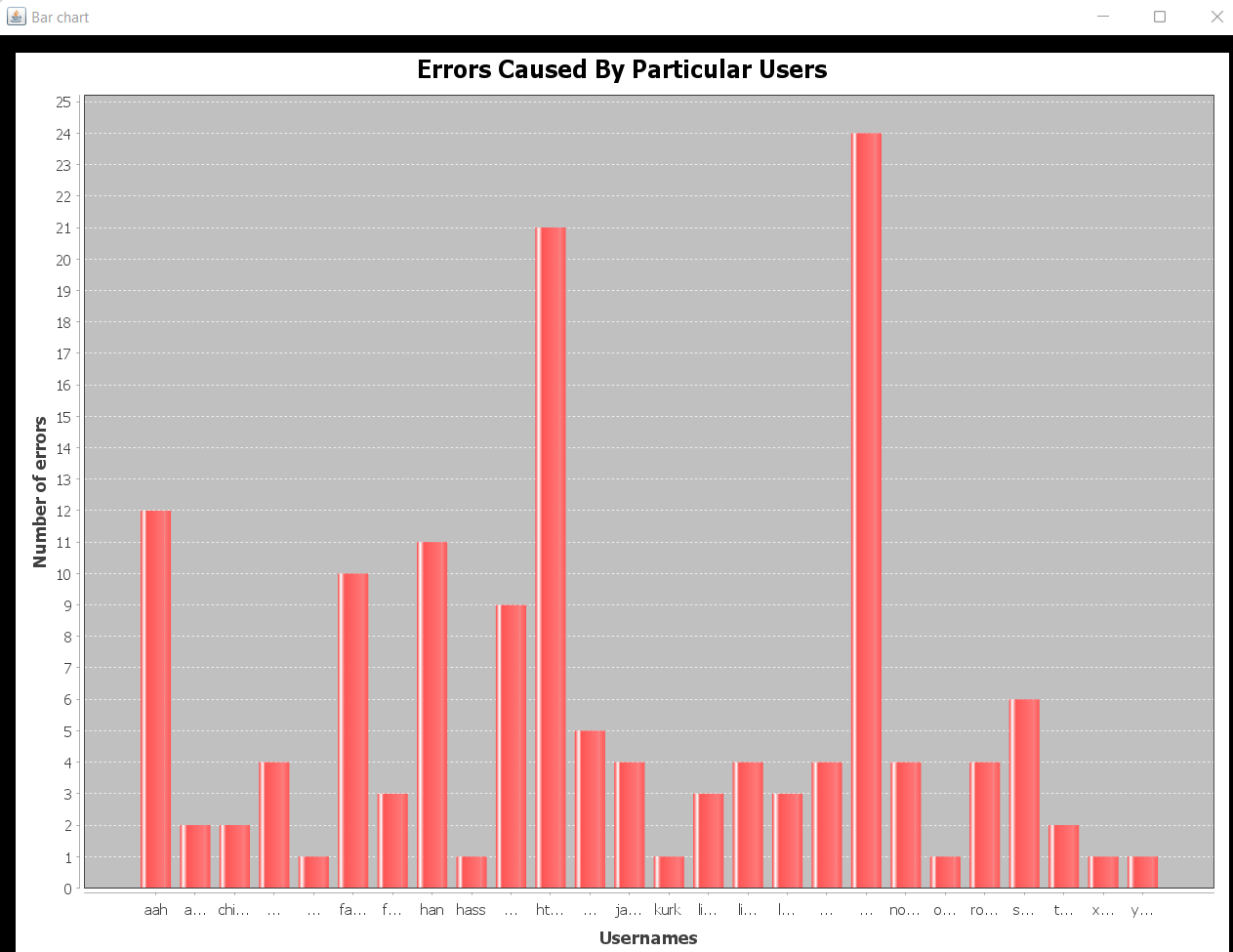
*\*As a result, the lines of errors and total number of errors within the range will be displayed at the console.*

**Graph/Chart:**

***6) Display a pie chart:***



***7) Display a bar chart:***



**Description:**

From the result above, there are few methods used to extract the data and information needed.

***Total number of errors:***

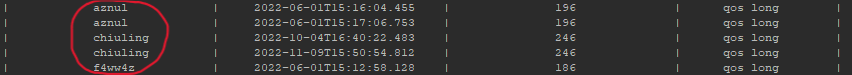
To get the total number of errors caused by users, a method named ***countErrors()*** is used. This method provides the function reading a text file that contains raw data and counts the number of lines containing errors caused by the user. The total number of errors is then being returned.

***Table with detail informations:***

There are two methods used to display these 2 tables as there are two types of table being able to displayed:

***1) Table with name sorted by alphabet:***

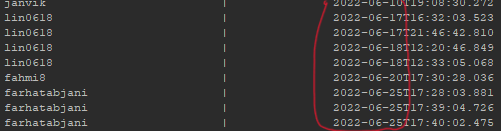
Eg:



First method used is ***displayInfoInTable(HashSet<String> users)***. This method takes in a HashSet of usernames as a parameter and displays the data in a table format. (The HashSet which stores usernames is returned from another function called ***findAndInputUsernames()***.) The function first converts the HashSet into an ArrayList and sorts it so that the usernames are arranged according to the alphabet. Then it reads a file called **"errors.txt"** and for each username in the ArrayList, it reads the file line by line and looks for lines that contain the username and match a specific pattern using a regular expression. If a line matches the pattern, it extracts specific information from that line and prints it in a table format using the printf method. The table includes columns for the name, time, number of associations, and error types.

***2) Table with information arranged according to time:***

Eg:



Another method used is ***displayInfoInTable(ArrayList<String> months).*** (The HashSet which stores months(06, 08, 09, 10, 11) contained in lines of errors is returned from another method called ***sortErrorsByMonths()***).This method takes in an ArrayList of months as a parameter and displays the data in a table format. This method reads the file **"errors.txt"** and for each line and extracts specific information from that line such as the username, time, number of associations, and error types and prints it in a table format using the printf method. The table includes columns for the name, time, number of associations, and error types.

Both of these functions used **regular expressions** which in short term is ***regex***. ***Regex*** is implemented through the use of the **java.util.regex package**. This package contains classes that provide support for regular expressions, including the **Pattern** and **Matcher** classes. A specific **Pattern** is defined so that when the **BufferedReader** reads each line of text, it uses Matcher to check whether the line matches the defined pattern or not. If matches, then some of the data is extracted from that line of text by using **matcher.group(int)**.

***Username list and number of errors caused by users:***

To display a username list with the number of errors caused by that particular user, a method named ***countErrorsCausedByUsers(HashSet<String> users)*** is used. This method takes in a HashSet of usernames as a parameter and counts the number of errors caused by each user. The method first converts the HashSet into an ArrayList, sorts it so the usernames in the HashSet are arranged according to the alphabet, then it reads a file called **"usernames.txt"**. For each username in the ArrayList, it reads the file line by line and looks for lines that contain the username. It then increments a counter for each line that contains the username. This process will repeat until all usernames in the ArrayList are iterated.

(The HashSet which stores usernames is returned from a method called ***findAndInputUsernames()****.* This method also writes the usernames into a text file called **“usernames.txt”**.)

***Count errors in a given range of months:***

To count the number of errors within two months, a method called ***countErrorsWithinMonths(int startMonth, int endMonth)*** is used.

This method, ***countErrorsWithinMonths(int startMonth, int endMonth)***, takes in two integer parameters, **startMonth** and **endMonth** as a range, and counts the number of errors that occurred within this given range inputted by user. This method reads a file named **"errors.txt**" and for each line, it looks for lines that match a specific pattern using a regular expression. The regular expression extracts the month from the date. If a line matches the pattern and the extracted month is between the startMonth and endMonth (inclusive), the function increments a counter. It then returns the count of errors that occurred within the given range of months.

While a method called ***IsInteger(String startMonth, String endMonth)*** is used to check whether the **startMonth** and **endMonth** input by user in **Main** is an integer or not.

***Find errors in a given range of date:***

To find the errors from a specific date to another, a method called ***countErrorsBetween(String start, String end).***

This method, ***countErrorsBetween(String start, String end)***, takes in two String parameters, **start** and **end**, which represent the start and end dates for the range. The method reads a file named **"errorsByDates.txt"** which contains dates in the format **"yyyy-MM-dd"** and for each line, it creates a LocalDate object from the date in the file using the LocalDate.parse() method.

Then it compares the date from the file with the start and end dates passed as parameters. If the date from the file is between the start and end dates (inclusive), the function increments a counter and prints the line.

Finally, it prints the total number of errors that occurred within the given range of dates.

The text file **"errorsByDates.txt"** is created by another method called ***getDateOfErrors()***. This method, ***getDateOfErrors()***, reads a file named **"errors.txt"** and for each line, it looks for lines that match a specific pattern using a regular expression. The regular expression is used to extract the date, association number, username and error type from the line. If a line matches the pattern, it adds the date to a HashSet of dates, and writes the extracted information to a file named "**errorsByDates.txt'**' in the format "**yyyy-MM-dd** **association\_number** **username error\_type**" .

It then returns the HashSet of dates.

While another method named ***isValidDateFormat(String date)*** is used to check the format of date input by user in ***Main*** is valid or not. This method uses the DateTimeFormatter class to define the expected pattern, and then tries to parse the input String **'date'** into a LocalDate object using that pattern. If the parse is successful, it returns true, indicating that the input String is in a valid format. If an exception is thrown during the parse, it returns false, indicating that the input String is not in a valid format.

***Others:***

Another method which is named ***FindErrorsInEachMonth(HashSet<String> months)*** is used to find the number of errors caused by users in each month and input them into a new file called **“NumErrorsEachMonths.txt”.**

This method takes in a HashSet of Strings called "**months**" and finds errors in files named "**month[month].txt**" where [month] is one of the elements in the HashSet "months". An integer variable ‘count’ is defined with an initial value of zero.

The method creates an ArrayList called "month" and copies all elements from the input HashSet "months" into it.(The HashSet which stores months(06, 08, 09, 10, 11) contained in lines of errors is returned from another method called ***sortErrorsByMonths()***). The ArrayList is then sorted.

For each iteration of elements in the ArrayList, the method opens a file named **"C:\Users\User\Desktop\FOP\_Assignment\yap\month[element].txt"** and reads it line by line. It uses a regular expression pattern "**\[.-([element])-[0-9]{2}.\].(error: This association).**" to match lines that contain the error "error: This association" in the month specified by the current element in the ArrayList. When a match is found, the count variable is incremented.

Finally, the method writes the current element(month) and count variable in the **“NumErrorsEachMonths.txt”** file.

All the “**month[element].txt**" text files stated above are written by using a method called ***inputErrorsByMonths(HashSet<String> months)***. This method takes in a HashSet of strings called "months" and processes a file named **“errors.txt”**. The method creates an ArrayList called "month" and copies all elements from the input HashSet "months" into it. The ArrayList is then sorted.

For each iteration of elements in the ArrayList, the method opens a file named "C:\Users\User\Desktop\FOP\_Assignment\yap\month[element].txt" and reads the text file **“errors.txt”** line by line. It uses a regular expression pattern **"\[.-(" + month.get(i) + ")-[0-9]{2}.\].(error: This association).**" to match lines that contain the error "error: This association" in the month specified by the current element in the ArrayList. When a match is found, the line is written to the **month[element].txt** file.

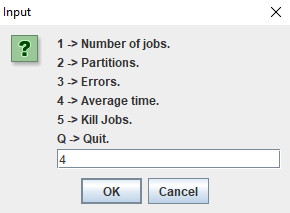
The data stored in the **“NumErrorsEachMonths.txt”** will then be used as the data information being displayed in the pie chart and bar chart which are shown at the section Sample Output above.

**(D) Average execution time of the jobs submitted, Execution time of jobs submitted based on month, Average execution time of the jobs submitted based on month , job with longest runtime every month, job with shortest runtime per month,**

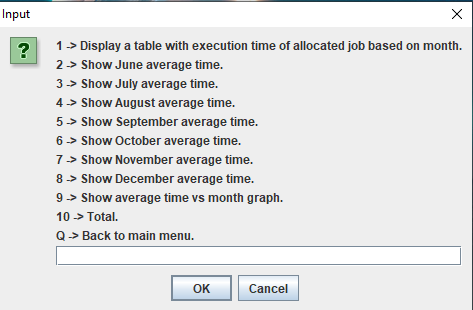
**Source code ( Input link) :**

[Part (D) source code](https://docs.google.com/document/d/1CSIbqqYinum1dmgjCRYhEYKnFmP3hhEbnonqE4osE5s/edit?usp=sharing)

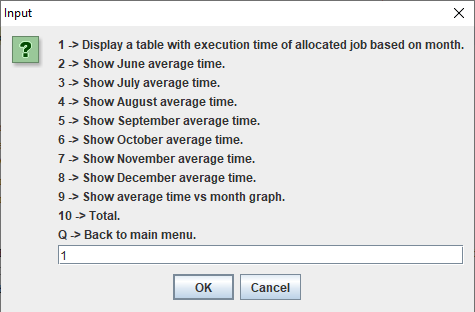
**Sample Output:**



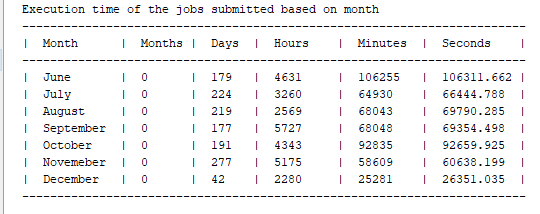
***Users are prompted to enter number 4 to get into the Average time part.***

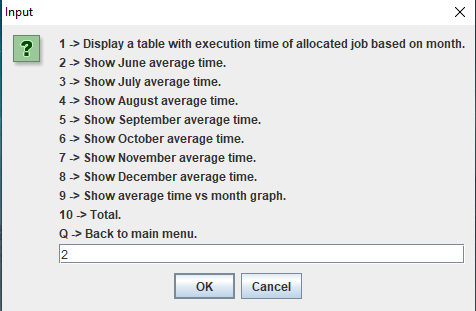


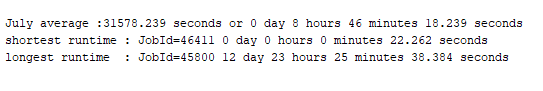
***After getting into the Kill Jobs part, there are a few options here and you are free to enter any numbers from 1 to 10 to observe and display the outputs that you want****.*

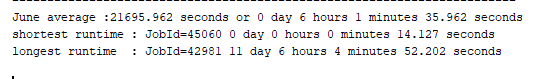
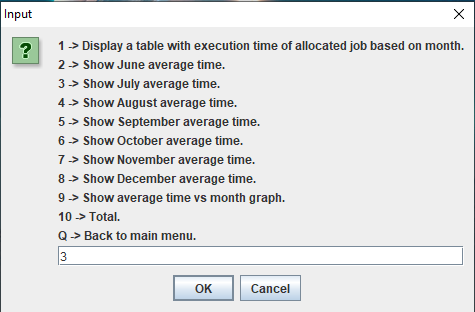


***Enter number 1 to display the table for table with execution time of allocated job based on month***

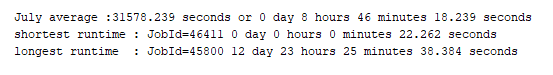
***A table will be printed out in the terminal of the IDE***

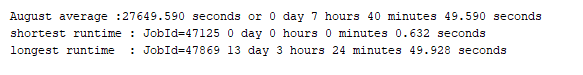
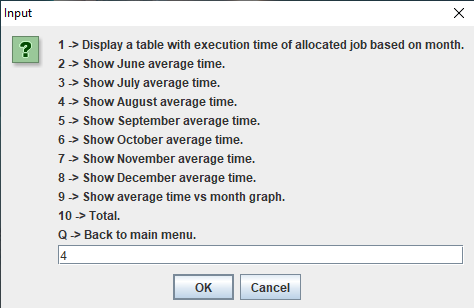


***Enter number 2 to show average time,shortest and longest runtime of June***

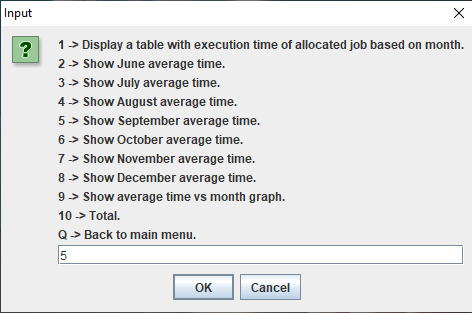
***The result will be displayed on the terminal of the IDE***

***Enter number 3 to show average time,shortest and longest runtime of July.***

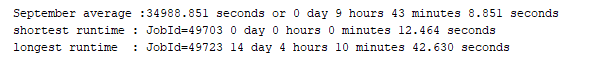
***The result will be displayed on the terminal of the IDE.***

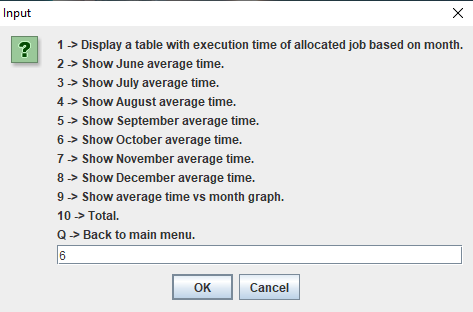
***Enter number 4 to show average time,shortest and longest runtime of August***

***The result will be displayed on the terminal of the IDE.***

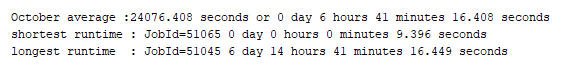


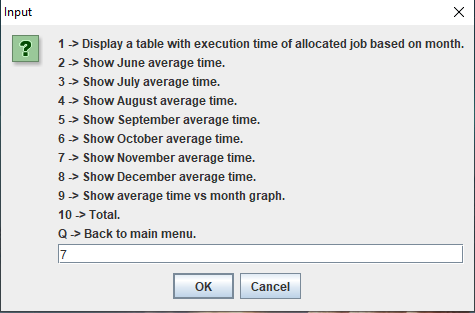
***Enter number 5 to show average time,shortest and longest runtime of September.***

***The result will be displayed on the terminal of the IDE****.*

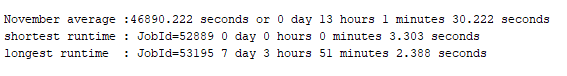


***Enter number 6 the table to show average time,shortest and longest runtime of October***

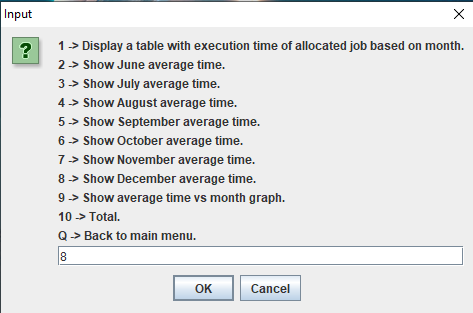
***The result will be displayed on the terminal of the IDE.***



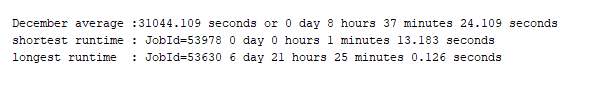
***Enter number 7 to show average time,shortest and longest runtime of November***

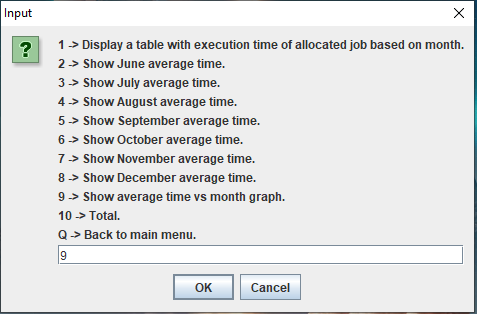


***The result will be displayed on the terminal of the IDE.***

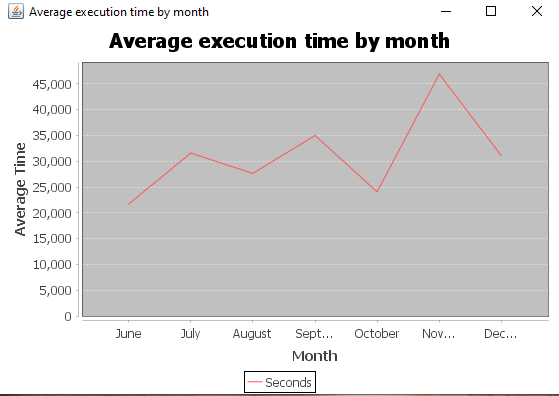


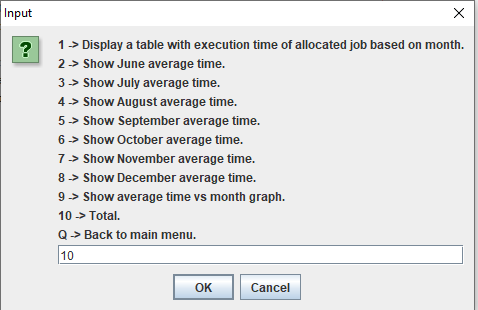
***Enter number 8 to show average time,shortest and longest runtime of December***



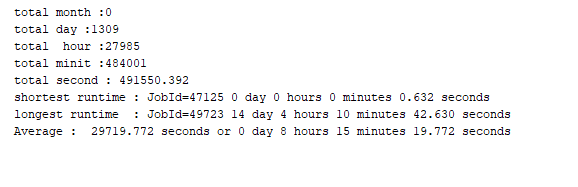
***The result will be displayed on the terminal of the IDE.***

***Enter number 9 to display the line chart for average time of allocated job based on month***



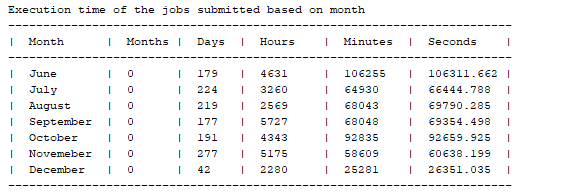
***A line chart window will pop up.***

***Enter number 10 to show the total month,day,hour,minutes,seconds and also the shortest and longest runtime of jobs for all months.***



***The result will be displayed on the terminal of the IDE.***

**Table:**



***Table of execution time of the jobs submitted based on month***

**Graph/Chart:** 

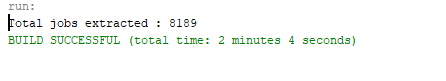
***Graph of average time vs month***

**Description:**

**Before the main:**

Before starting to extract the time, firstly another program is written to extract the allocated job based and the job complete line from the log file to be written to 2 separate files.Firstly, we used an *arraylist* to store each line read from the log file.The *pattern* class is used to find the line from the log file that contains **“Allocated JobId”** or  **“sched/backfill:”** inside a *for* loop **.** If the line is found then we use another *pattern* class to find the line that contains **“\_job\_complete:”** inside another *for l*oop**.** If the line is found, a *split* method is used to get the job id for both the **“Allocated JobId”** or  **“sched/backfill:”** line and the **“\_job\_complete:”** line. If both job id of the line matches and the word “done” is found in the **“\_job\_complete:”** line, the program will write the **“Allocated JobId”** or  **“sched/backfill:”** line into a text file named **“jobcreated.txt”**, and the line **“\_job\_complete:”**  will be written into another file named **“jobdone.txt”.** A counter is also used to count the total jobs extracted .

**The reason this program is written in another class is because the time required to completely extract all the lines(compilation time) is quite long, thus we decide to separate it from the main.**

****

**Source code :** [Source code to get job created and job done](https://docs.google.com/document/d/179a1EiYj9BfpXsODjAAF2_jpyD_fa_R9hZMOdWFurhE/edit?usp=sharing)

**Methods used**

**public static double gettotal(int day,int hour,int minit,double sec)**

* This method will convert all types of time into seconds first. Then it will total it up all together and return it back.

**public static void stos(double totalseconds)**

* This method will receive the total seconds and convert it into number of days,hours, minutes and seconds. Then the method will print the converted data in the formate of “ xx day xx hour xx minutes xx seconds (in 3 decimal value) “

**public static double getavg(int counter,double second,int minutes,int days,int hour)**

* This method will convert the second, minutes, days and hour into seconds then total it up with the input seconds . Then the total seconds will be divided by the number of jobs (counter) and the average value will be returned.

**public static double getsecond(double a ,double b)**

* This method is used to find the number of seconds used to execute a job.The method will receive the seconds from the date of job created and job done.Then 60 second will minusthe second of job created to get the execution time. The execution time will added with the second of the job done and the total value will be returned back.

**public static int getminutes(int a, int b)**

* This method is used to find the number of seconds used to execute a job. The method will receive the minutes from the date of job creation and job done. 60 minutes will be minus the minutes of the job created to get the execution time. The execution time will be added with the minutes of the job done . If the total value is greater than zero it will return the value minusing 1. This is because we actually start counting from the next minute as most of the time does not start exactly from 0 seconds and the **getsecond** method already helps us count the seconds until the next minutes.

**public static int gethour(int a,int b,int c)**

* This method is used to find the number of hours used to execute a job. This method will receive the hours from the date of job created and job done.The method will also receive the day difference which means the date of the day the job done minus job created. If the difference is more than zero which means its execution time exceeded a day, 24 hours will minus the hour of the job created . Then the value will be total up with the hour of job done. Else, if the execution time is within the same day , the total hour executed can obtain by just minusing the hour of job done wil job created. Then, if the total hour is greater than zero, 1 hour will be deducted before returning it . This is because we actually start counting from the next hour as most of the time does not start exactly from 0 minutes and the **getminutes** method already helps us count the seconds until the next hour.

**public static int getday(int a,int b,int c,int d,int e)**

* This method is used to calculate the number of days used to execute a job. This method will receive the day from the date of job created and job done. The method will also receive the month difference and the month from the job created and job done . If month difference is one and if the month created is june,september, november, 30 days will minus the day of the date from job created else if the month created is july,august,october,december, 31 days will minus the day to get the execution days.Then the value will be added with the day of the date from job done to get total day. Else if the month difference is more than or equal to 2, we use the same method above to get the day of the job created based on month. Then, a for loop is used to get the day for the whole month and will be total up with the day of job created. Then, the new total value will be total up with the day of the date of job done. If the month difference is zero, the total days will be the day of job done minus day of job created. Before returning the total value, if the total value more than one, 1 day will be deducted from the total value and then it will be return to the user.

**public static int getmonth(int a,int b)**

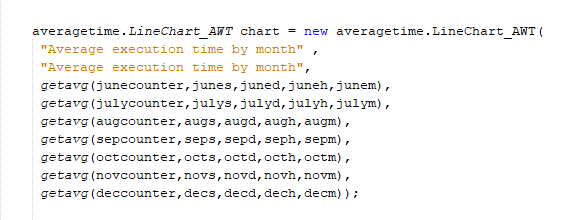
* This method is used to get the month. The method receive the month created and month done. Then if month done minus month created less than or equal one, the method will return zero. Else, the method will return month done minus month created minus 1.

**Class used :**

**public static class LineChart\_AWT extends ApplicationFrame**

* The class is used to print out the line chart for the average time based on month. Inside the class, there are methods to receive the average time for each month. So we can just put in the **getavg** method with variables of each month of the method in the main and the graph will be printed out with the value of each month.

Exp:



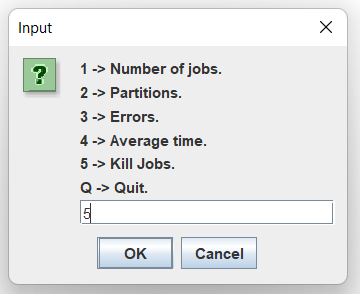
**Main :**

Buffered reader class is used to read both the **“jobcreated.txt”** and **“jobdone.txt”** files. All the variables used are first declared. A *for* loop is used to loop through each line of the text inside both the text file. Inside the for loop, firstly *split* class is used to obtain the month,day,hour,minute and second of the job allocated and the job done, and each type of time is initialized into a variable. Then, the number of seconds used to complete the job is obtained using the **getsecond** method. This is also the same for the minutes,hour,day and month too, using **getminutes**,**gethour**,**getday**, and **getmonth** method. Then, there is a counter to total up each type of time for every job .Then, a *if* statement containing the method **gettotal** is used to compare with another variable is used to obtain the shortest/minimum execution time for the job . If the method **gettotal** is less than the variable junemintotal,the variable will change to the value of the **gettotal** method and the id of the job will be collected and initialized into a variable. For the maximum/longest execution time, it will be the opposite of the minimum using its own variables.Then, a *if* statement is used to detect the month of the job .Every month available in the log file has its own statement. Inside the *if* statement each type of time has a variable to total up each type of execution time for the job allocated within the month. Then the maximum and minimum execution time of the job allocated within the month will be obtained using the exact same way like above. The *for* loops ends, and the data collected for each month is displayed in a table format. Then, the average time , maximum and minimum execution time for the job allocated in each particular month is also displayed. The **getaverage** and **stos** method are used to get the average time and display the time. Then, a line chart of the average execution time of the job allocated in a particular month is displayed by calling the line chart method using the data obtained above.

**(E) Number of jobs successfully being killed and being returned (extra feature)**

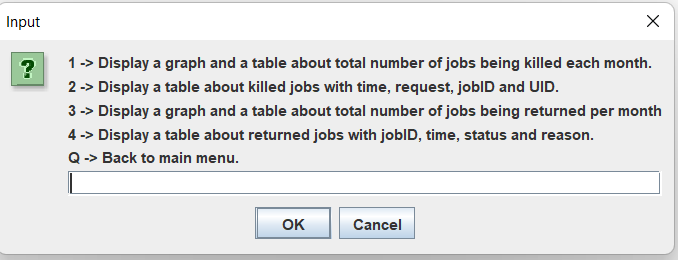
**Source Code (Input link):** [Source Code for Part E (FOP ASSIGNMENT).docx](https://docs.google.com/document/d/1lUntZQHSVBFKc_8IG3Kcue73Ry2PXQbb/edit?usp=share_link&ouid=107376005665754289629&rtpof=true&sd=true)

**Sample Output:**



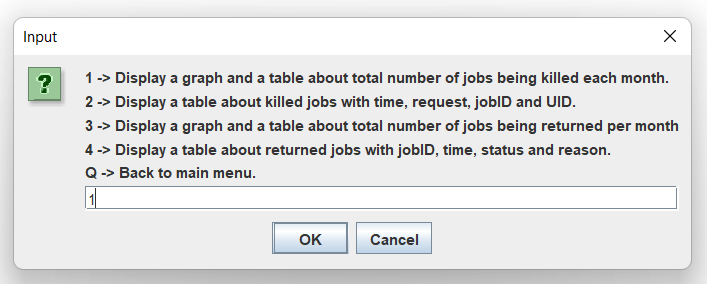
*Users are prompted to enter number 5 to get*

*into the Kill Jobs part.*

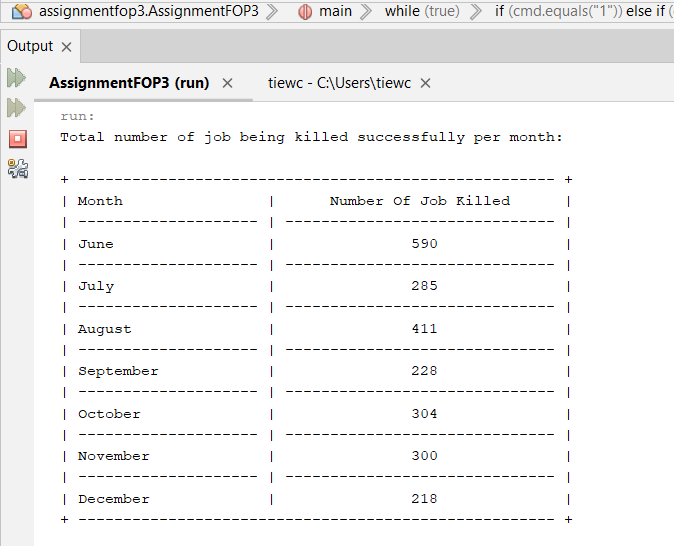


*After getting into the Kill Jobs part, there are a few options here and you are free to enter any numbers from 1 to 4 to observe and display the outputs that you want.*

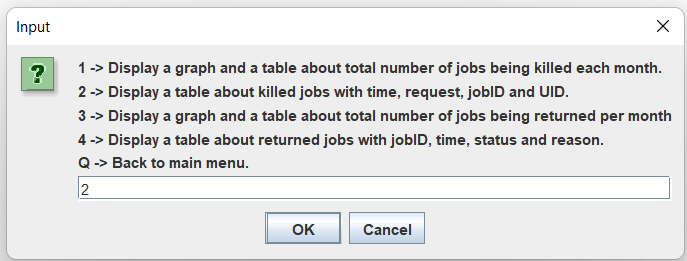
**Table:**



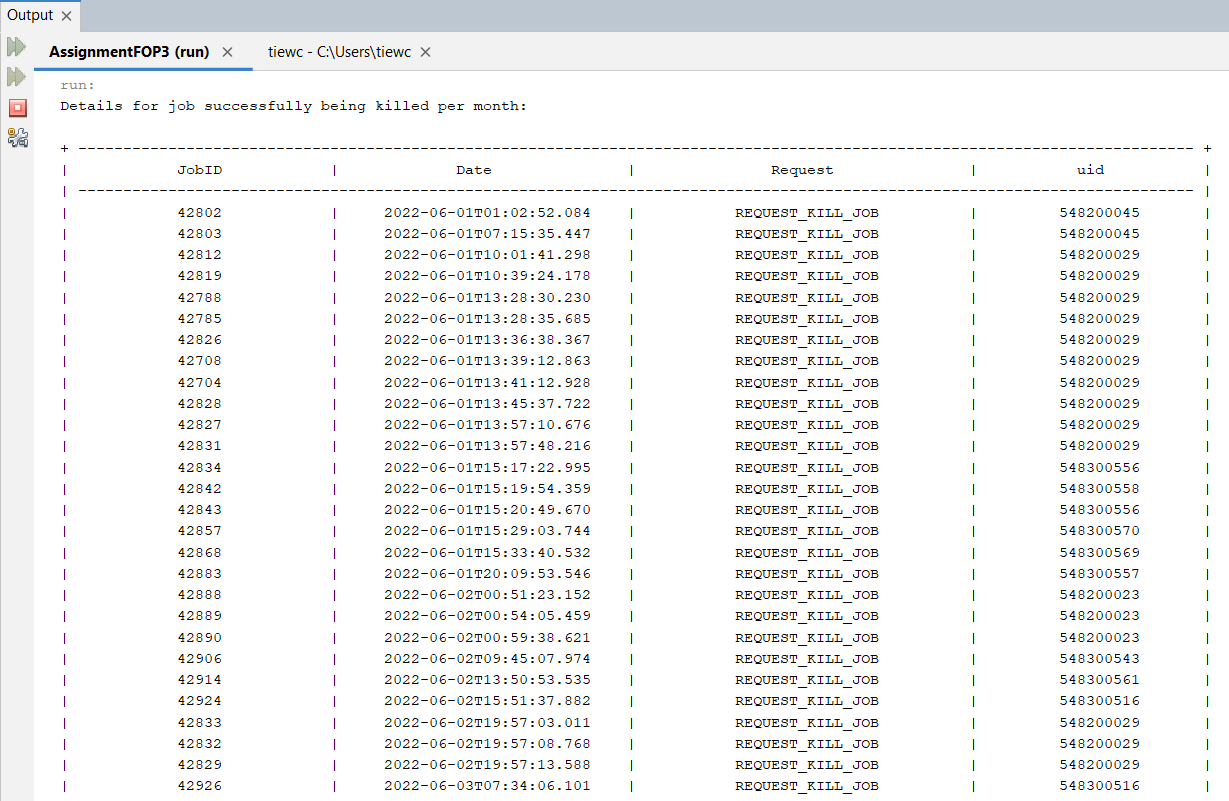
*GUI: Enter number 1 to display the table for total number of job being killed successfully*

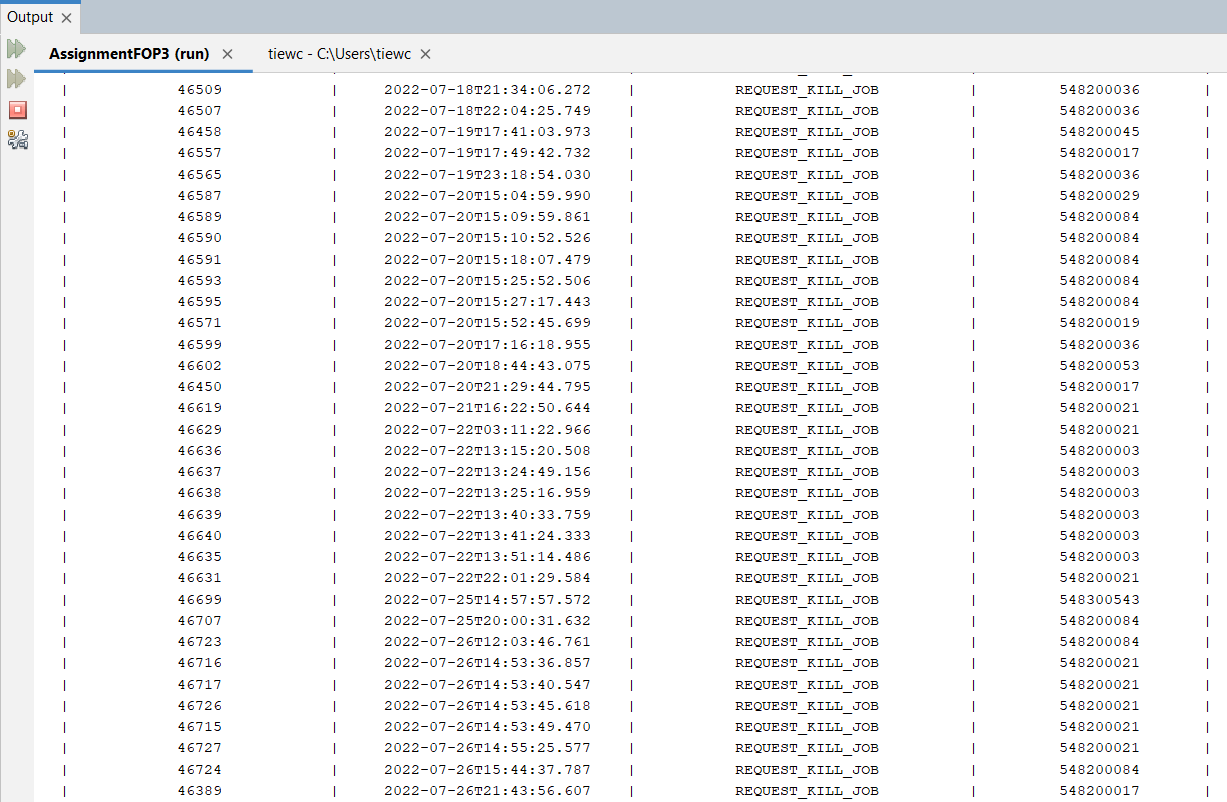


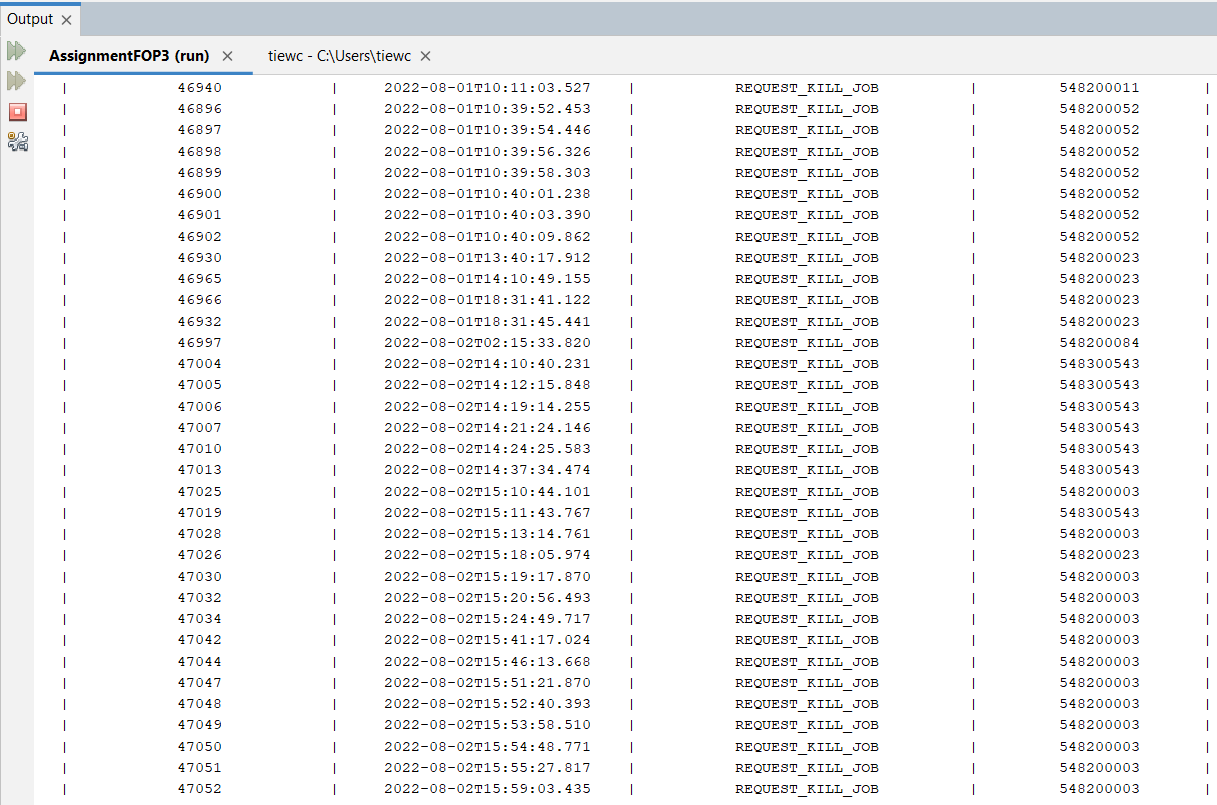
*Table (E1) : Table to display the total number of job being killed successfully, all of the data is being sorted by months*

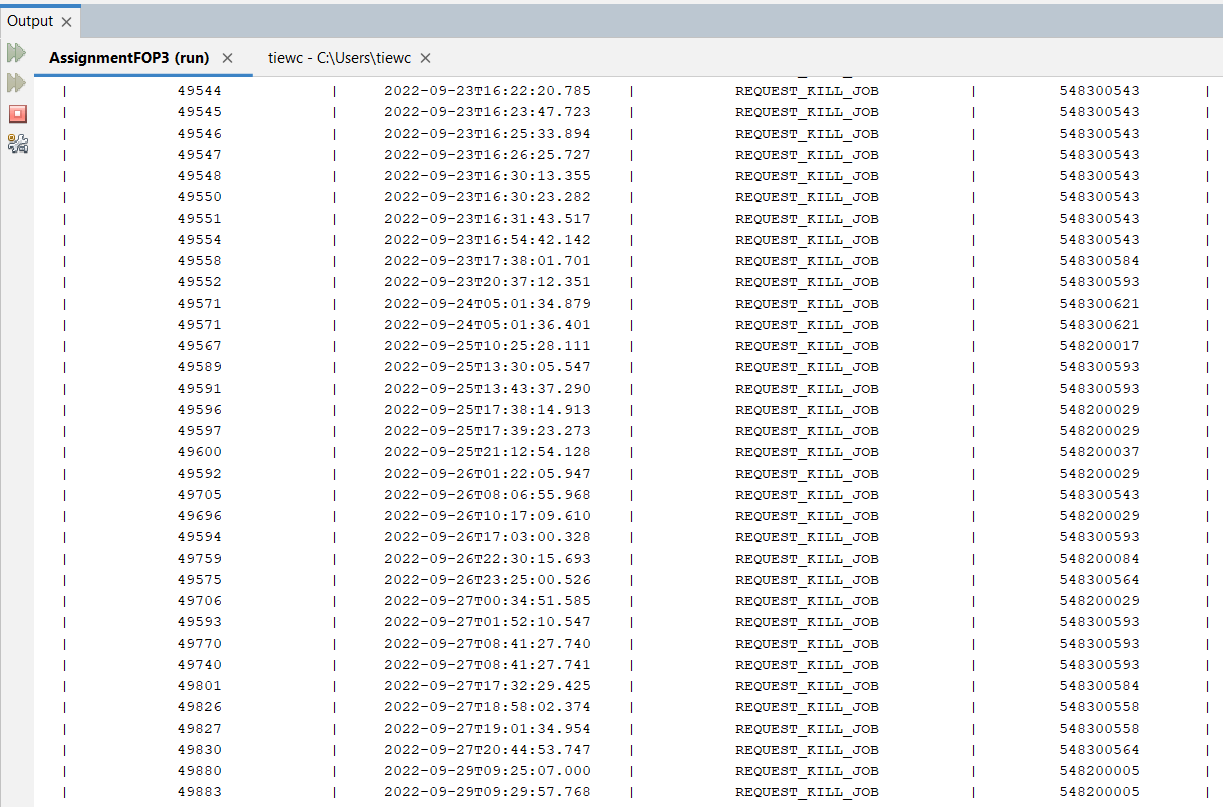


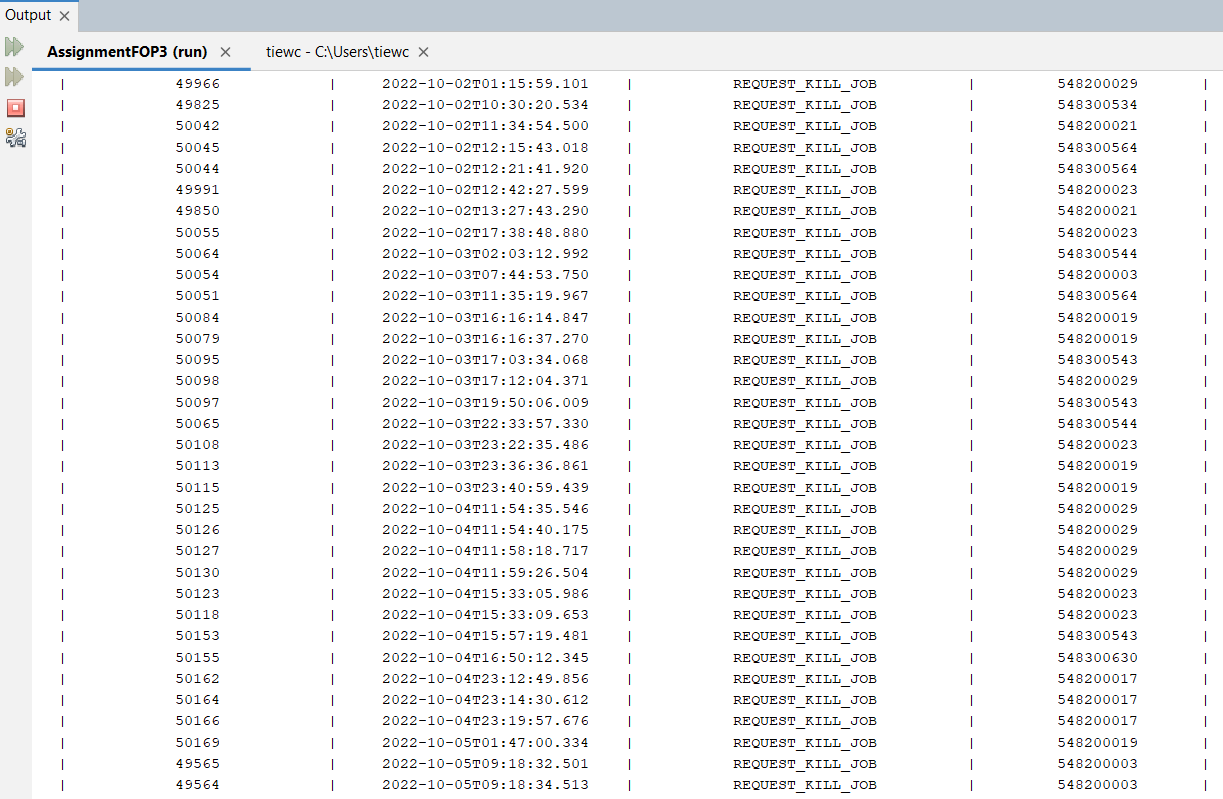
*GUI: Enter number 2 to display the table for details about the jobs being killed successfully*

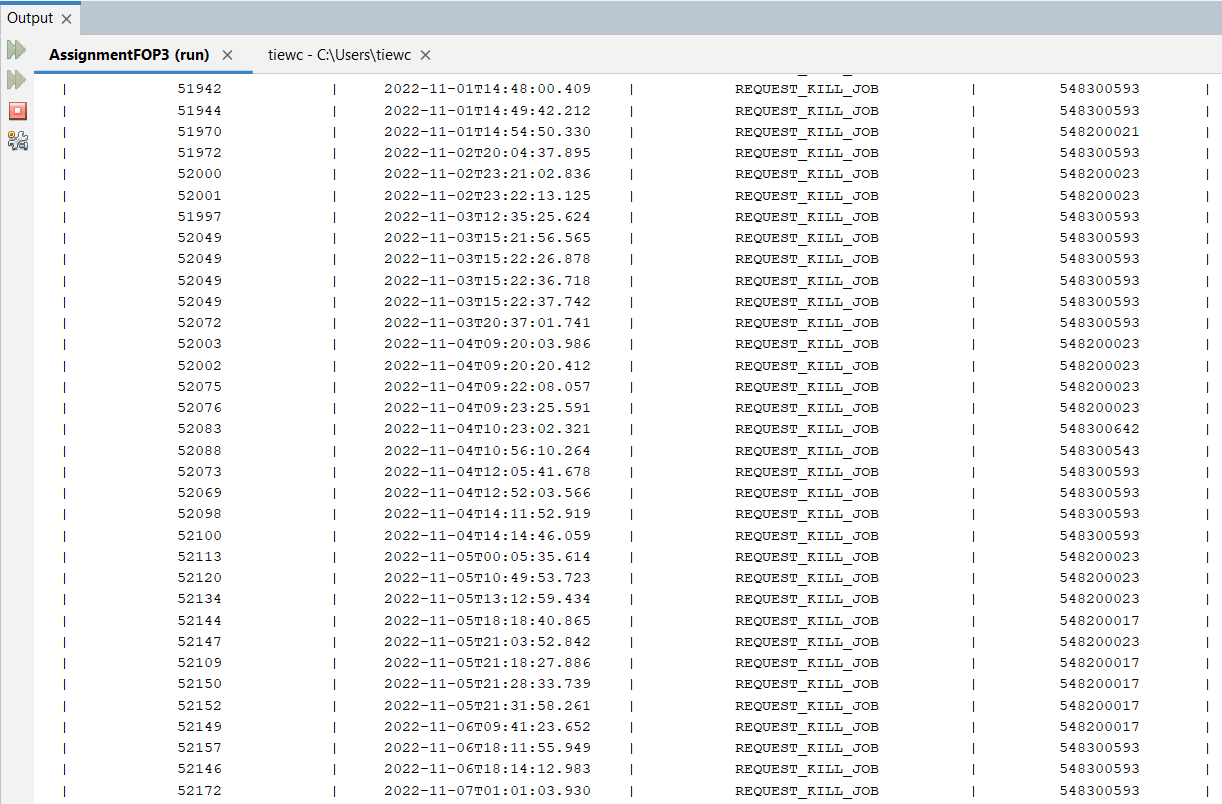


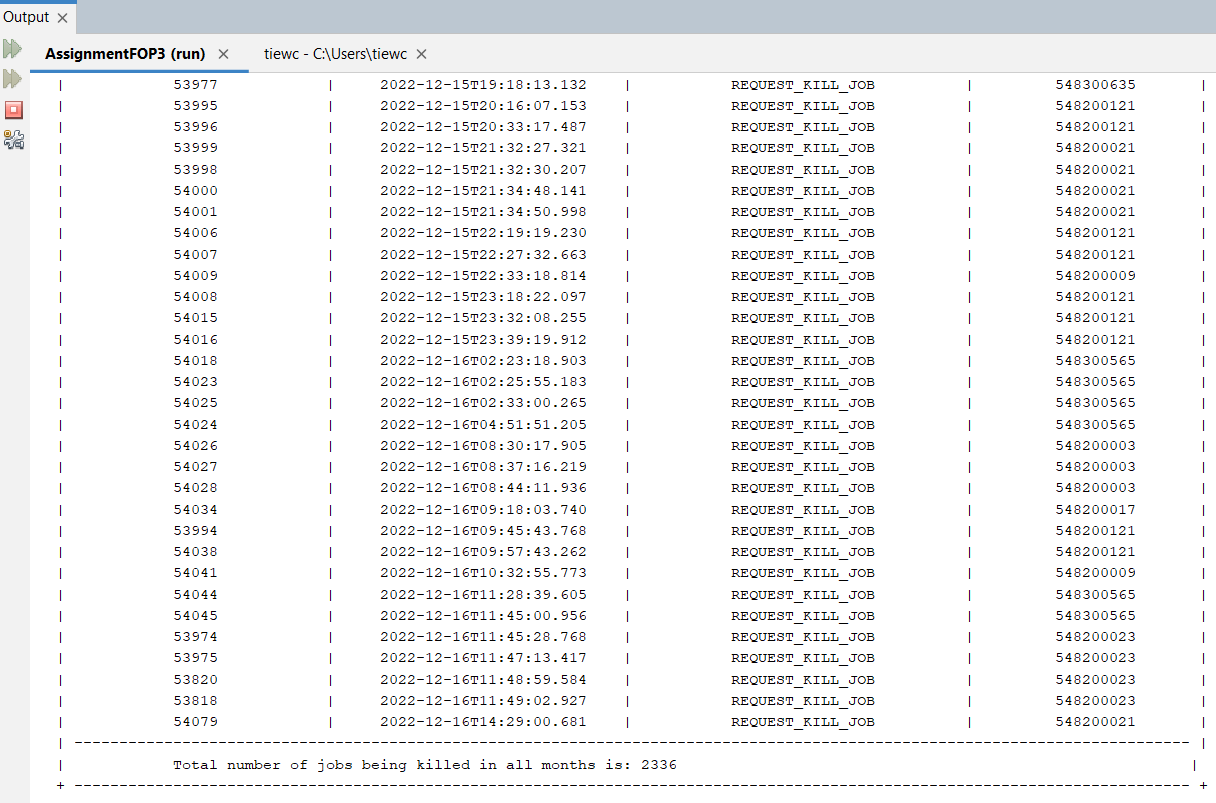




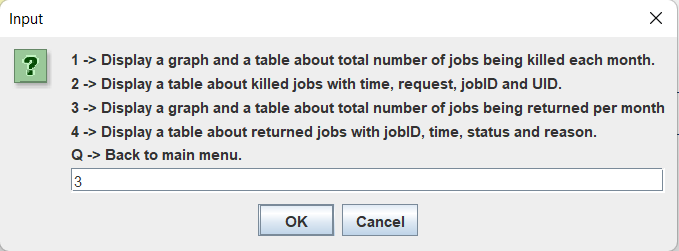
****

****

****

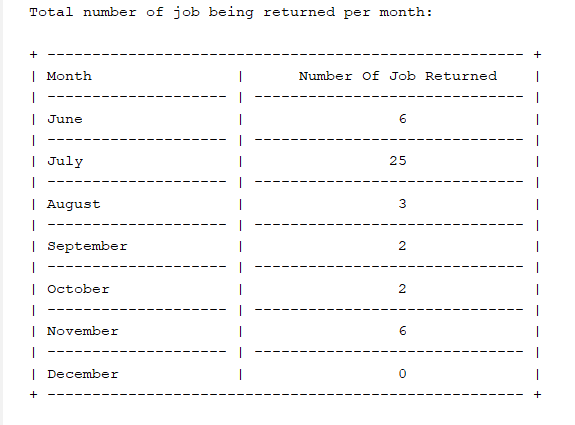
****

*Table (E1) : Table to display the details* ***(JobID, Date, Request, uid)*** *of all jobs successfully being killed from June to December*

****

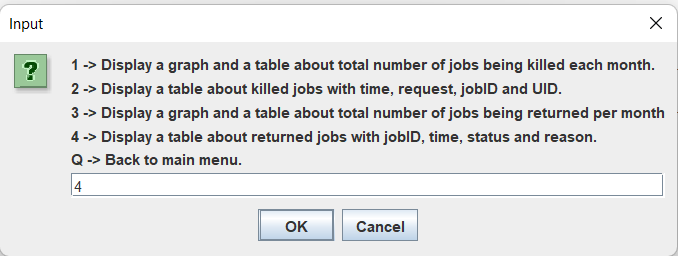
*GUI: Enter number 3 to display the table for total number of job*

*being returned*

****

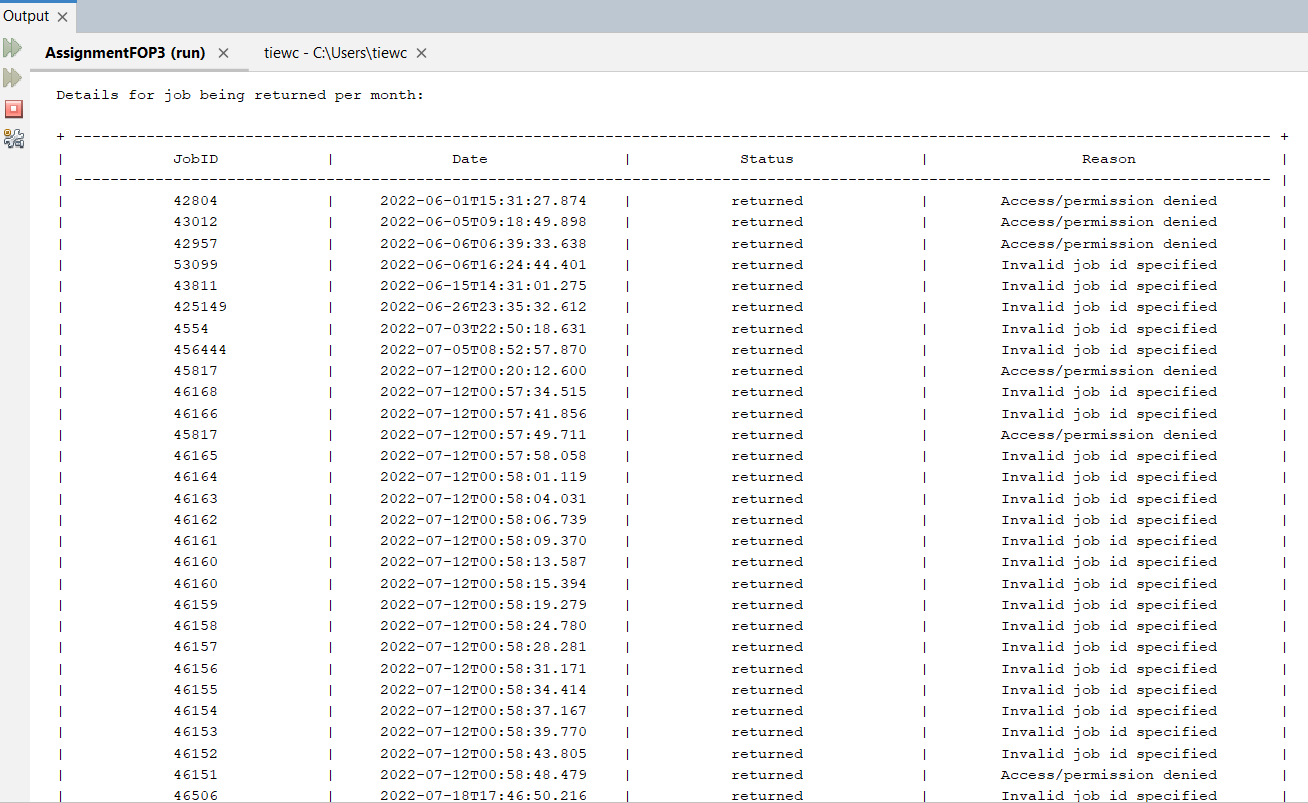
*Table (E2) : Table to display the total number of job being returned, all of*

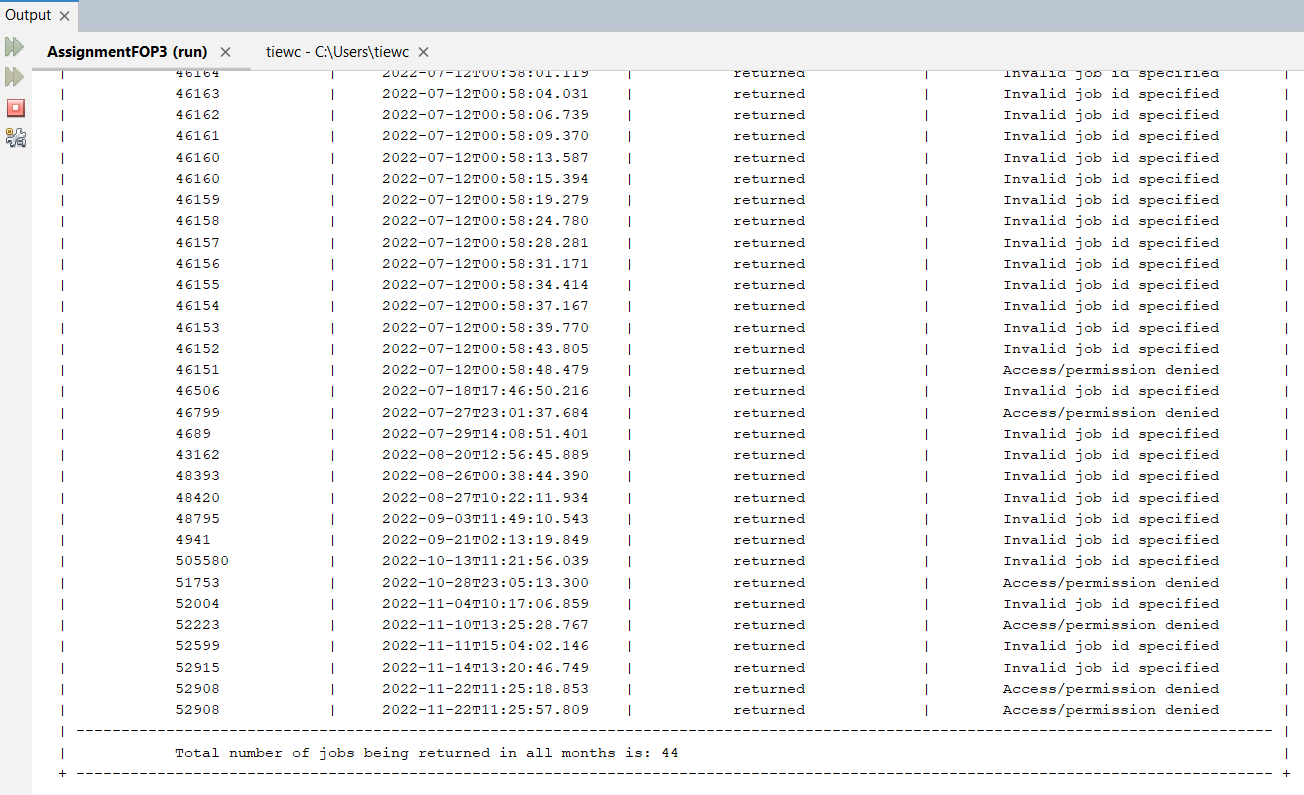
*the data is sorted by months*

****

*GUI: Enter number 4 to display the table for details about*

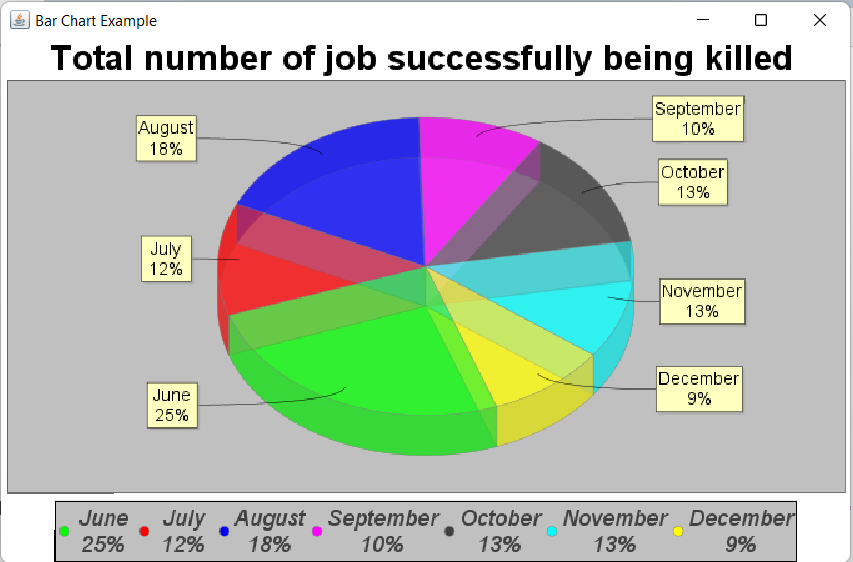
*the returned jobs*

****

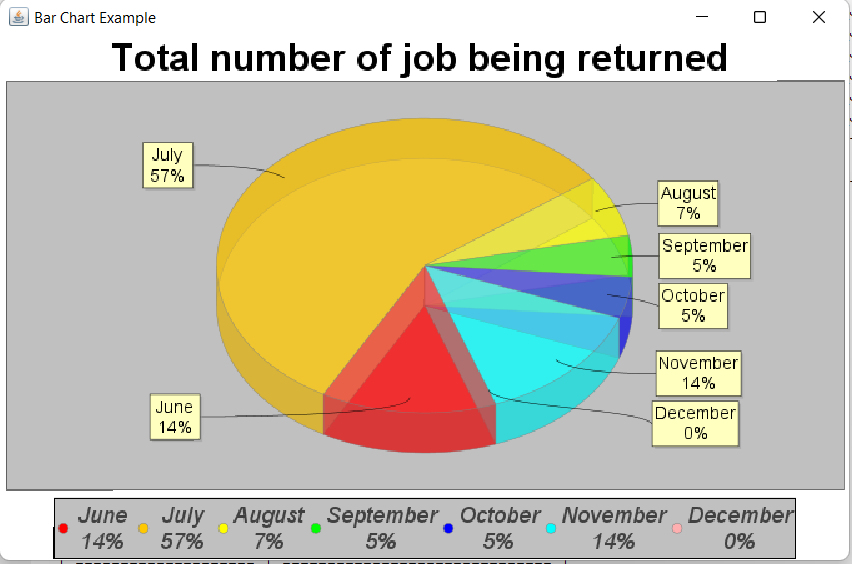
****

*Table (E2) : Table to display the details* ***(JobID, Date, Status, Reason)*** *of job returned from June to December*

**Graph/Chart:**



*Pie Chart (E1) : Total number of job successfully being killed from June to December*



*Pie Chart (E2) : Total number of job being returned from June to December*

**Description:**

**Part 1: Table and Pie Chart for total number of jobs successfully being killed:**

There is a method called **totalkilljobmonth()** to display a table and show the data for the total number of jobs being killed from June to December. Three arrays which are *successcount*, *requestcount* and *failurecount* will be assigned. A String array named *Month* will display the name of the months from June to December. Read the file named **“DEFAULT”** line after line by declaring a String variable called *line*. Check whether the line consists of **“\_slurm\_rpc\_kill\_job”** and **“REQUEST\_KILL\_JOB'**' by using an if-else statement, call method named *checkMonth()* passing *line* and *requestcount* array as arguments. If the passed line matches the pattern class, the elements inside the *requestcount* array will increase by 1. Else if the line contains **“job\_str\_signal()”**, it will pass to method *checkMonth()* with parameter *line* and *failurecount* array. If the passed line matches the pattern, the elements inside the *failurecount* array will increase by 1. In order to calculate the number of jobs successfully being killed for each month, each element in the *requestcount[i]* will be subtracted with elements in the *failurecount[i]* inside a for loop. The table then will be printed based on the information obtained. The *successcount* array will become the arguments for a method called **PieChartforsuccesskilljob()** to build the pie chart for it. Each of the elements in the *successcount* array represents the number of jobs successfully being killed in that particular month.

**Part 2 : Table for details of jobs successfully being killed:**

This **totalkilljob()** method is used to display the table for the details of jobs successfully being killed per month. Several arrays will be assigned as *successrequest* and *failrequest* to display jobid, date, request and uid of the users. After reading the file, two classes of patterns and matches will be defined to match the *line* from the input. If the *line* matches the first pattern class, each element for all *successreques*t arrays will increase by 1. When the *line* contains the **“job\_str\_signal()”**, and String *matcher.group(5)* in the second pattern class equals to id which is String *matcher.group(3)* in the first pattern class, (id in the current line is the same as the one in the previous line), the elements in all *successrequest* arrays will become null by decreasing the current index by 1. At the same time, all the elements in all *failrequest* arrays will increase by 1. A variable named *counter* will be declared to count for the total number of jobs successfully being killed. By using the for loop and printf statement, the user jobid, date, request and uid will be displayed in the table by using the respective elements in all *successrequest* arrays.

**Part 3 : Extract data for returned job:**

The **TotalReturnedJob()** method is used to extract all of the data consisting of **“(sig=9 returned)”** which are the errors and stream all of them into a file called **“D1”** by reading the data from the **“DEFAULT”** file.

**Part 4 : Table and Pie Chart for total number of jobs being returned:**

The **MonthReturnedJob()** method is used to display the table for the total number of jobs being returned from June to December. A String array named *Month* will display the name of the months from June to December. By using a for loop, we can set all elements in the *count* array as zero. After reading the file, the element in the *count* array will increase by 1 whenever the line(input) contains the respective month of the year. The code then uses printf statement to format and display the information in the table. The value of each element in the *count* array will be passed into the **PieChartforreturnedjob()** method as the arguments. Each of the elements in the *count* array represents the number of the job being returned in that particular month.

**Part 5 : Table for details of jobs being returned:**

There is a method called **displayReturnedMonthInTable()** that is used to display the table for all the details of jobs returned from June to December, **(JobID, Date, Status, Reason).** Method will read the data from a file named **“DEFAULT”** and use the **repeat.()** method to construct the table. By using the for loop, we can easily print out the details for users with the respective *matcher.group()* when the *line* matches the pattern class. The *count* variable that indicates the total number of jobs being returned will increase by 1 each time the loop repeats again.

**5.0 Problem & Solution**

| **Problem** | **Solution** |
| --- | --- |
| 1. Codes requires long time to run and execute result | Simplify and modify codes by removing the redundant part of codes. |
| 1. Lack of skills and experiences to master Java Language for coding of the program | Improve Java Language programming skills by learning through platforms such as Youtube, Google, Udemy, Coursera |
| 1. Too much time is used to explore new tools for data visualization | Be patient to visualize the data in other ways and boost up momentum |
| 1. Outputs did not meet the expectation | Recheck the codes and debugging |
| 1. Unexpected problems occurred when creating graph or chart | Searching for guidelines and assistance through platforms like StackOverFlow and Youtube |

**6.0 Other Problem Arise**

1. As there is a huge amount of raw data, the program needs more RAM and processor to run it. So the program may not be runnable if the computer has lower RAM and processor and the program consumes a lot to extract the code.
2. Due to the huge amount of data in the text file, it is difficult to extract all the data accurately without any mistakes and once the mistakes happen, it is difficult to find out the parts which have not been extracted.
3. Since the data has been executed, it does not mean that the results of extraction are the same as the actual result as they might have other possibilities such as a continuous of invalid request of kill jobs in a certain period that can affect our program to find out the correct result.
4. In the text file, there have many situations that might can be happened which we cannot be expected such as the the jobs created is at the date which not between the range but the date jobs ended is between the range, this makes calculate the average of execution time become more challenging and increase the difficulty.
5. The window that displays a graph cannot be closed after being commanded by the user to pop out by using the UI.
6. System of some parts lacks some functions. For example, the system which checks Number Of Jobs, when we input invalid month, no reminder is shown to the user. Yet, there is no quit function in the system.
7. Some of the graphs and charts are done by hard coding, which means that the data of the charts and graphs are inputted manually. Other efficient ways should be used to solve this problem.

**7.0 Extra Features created**

1. UI:

The final output that has been produced in this assignment is an user interface with a list of commands and this UI will respond differently with every single command inputted by the user.

The UI created is a simple and user-friendly interface that has been built using the **JOptionPane** class in Java. **JOptionPane** is a class that provides pre-built dialog boxes that can be used to display information or receive input from the user. To import the JOptionPane class, you need to include an import statement in your Java file. This statement tells the compiler to include the JOptionPane class from the javax.swing package in your program.

The UI is designed to respond respectively to commands that are listed in different command lists which will be displayed. When the user inputs a command, the system uses JOptionPane to display a dialog box that shows the results of the command. While in some commands, the result of that command is shown at the console/terminal. The system is programmed to recognize specific commands and execute the corresponding action.

One of the key features of this UI is its ability to recognize and respond to specific commands. The command list is pre-programmed with a set of commands that the system is able to understand and execute. This ensures that the user is able to interact with the system in a predictable and consistent manner. Thus, when the command is out of the command list, the UI will gently remind that the command is invalid and prompt the user to enter again.

Another important feature of this UI is its ability to handle invalid commands. If the user inputs a command that is not listed in the command list, the system will recognize that the command is invalid and use JOptionPane to display a dialog box that informs the user that the command is not valid. This is important as it allows the user to know what commands are available and what actions they can perform with the system.

In terms of design, the UI is simple and minimalistic. The use of JOptionPane dialog boxes allows for a clear and concise display of information, making the interface easy to navigate. The use of JOptionPane showInputDialog also allows the user to input the command.

Overall, this UI is a straightforward and efficient solution that is designed to respond to specific commands in the command list using JOptionPane. Its simple design and ability to handle invalid commands make it easy for users to interact and use the system.

1. Graph:

To create charts and graphs, our group has explored **JFreeChart**. **JFreeChart** is a popular open-source Java library for creating charts and graphs. It provides a wide range of chart types, including line charts, bar charts, pie charts, and scatter plots, among others.

The following steps have been done to create chart and graph:

**First, download and include the JFreeChart library in our project.** We have downloaded the library from the JFreeChart website and added it to our project's classpath. This step is necessary to ensure that the JFreeChart classes are available for use in your program.

**Secondly, we create a dataset for our chart or graph.** JFreeChart supports several types of datasets, such as XYDataset, CategoryDataset, and PieDataset, depending on the type of graph you want to create. The dataset is the data that will be used to create the chart or graph. BufferedReader is used in code while creating graphs and charts to read data that have been extracted to a text file and use the data as information to be displayed in graphs and charts.

**Third, create the chart object using the dataset.** JFreeChart provides several chart classes, such as Line Chart, Bar Chart, Pie Chart, etc, that can be used to create different types of charts and graphs. Thus, we can use the appropriate chart class based on the type of graph you want to create. We have displayed our data mostly in Pie Chart and Bar Chart.

**Next, customize the chart.** Once we have created the chart object, we have set properties such as the title, axis labels, and legend. We can also add additional elements such as annotations and markers.

**Then, create a chart panel to display the chart.** The ChartPanel class provides a container for displaying a chart. Then we add the chart panel to our application's GUI to display the chart or graph.

**The following step is to add the chart panel to our application's GUI.** We can add the chart panel to a JFrame, JPanel, or other container.

After creating graphs and charts, we have coded some commands so that when the user input these specific commands, different graphs and charts will be displayed.

**8.0 Conclusion**

In this assignment, we have extracted the data from a huge amount of data in the log file. We have separated it into five parts , including number of jobs created/ended, number of jobs by partitions , numbers of errors, number of kill jobs and the average execution time of the jobs submitted. For the number of jobs created and ended, the data is extracted and shown by month or by day. For the number of jobs by partition, the data is extracted and shown by month and users are able to further know the date, time, and job id when a particular partition is being used. For the number of errors, users can find the total errors in a given range, the errors caused by users and the type of users caused by users. Furthermore,the data of the number of kills jobs is extracted for finding the total number of kill requests successful for each month and the date, time, job id and uid of every successful kill request. Finally, the data extracted for the average execution time is shown in the table, and chart.