Computer Science

COMPSCI 130

Assignment ONE

Due Date

Due: **11:59 pm Tuesday 17th September 2019**
Worth: **10% of the final mark**

Introduction

You have used Python lists and dictionaries on this real life research problem in Lab04. We will look at the same problem in this assignment but complete it in a different way. We would like to get a graphical display of how much data comes from a particular source host over time of an experiment. This is where your project comes in: You are to build an application that displays this data.

The total size of a trace file can vary substantially depending on the number of hosts involved in an experiment and the length of the experiment. Each line consists of a number of fields separated by a single tab character. Note that fields may be empty, in which case you get two successive tab characters.

The *first* field on the left is just a sequential number for each packet that is added by the eavesdropping program. The **second** (index 1) field is a **time stamp** that is also added by the eavesdropping program. Each trace file starts with a time stamp of 0.000000000 for the first packet in the file. The **THIRD** (index 2) field in each line is the IP address of the source host. IP addresses identify machines on the network and help routers forward packets between source and destination. Each IP address consists of four decimal numbers between 0 and 255 separated by dots (full stops). Fields seven, eight and nine are packet sizes in bytes. The size we're interested in here is that in field **EIGHT**, it's the IP packet size.

You only need to look at four fields: time stamp, source IP addresses, and IP packet size. Note that you may assume that all packets are valid in this assignment. An example is given as below:

43	33.835924000	192.168.0.10	8000	10.0.0.6	43032	74	60	0	1	1	0	0	0	1 1
44	33.851517000	192.168.0.10	8000	10.0.0.3	39548	74	60	0	1	1	0	0	0	1 1
45	33.866449000	192.168.0.11	8000	10.0.0.4	42899	74	60	0	1	1	0	0	0	1 1
46	33.938752000	192.168.0.15	8000	10.0.0.5	47745	1514	1500	1448	0	1	0	0	1	1 1
47	33.938766000	192.168.0.15	8000	10.0.0.5	47745	78	64	12	0	1	0	0	1449	1 1
48	33.939042000	192.168.0.15	8000	10.0.0.5	47745	726	712	660	0	1	1	0	1461	1 1
49	34.019781000	192.168.0.17	8000	10.0.0.10	55830	1514	1500	1448	0	1	0	0	1	1 1

Getting Started

The program consists of FIVE classes: `Table`, `Histogram`, `IP_address`, `A1`, and `A1GUI` You are required to complete the first FOUR classes in CodeRunner. Submit all classes to the web assignment dropbox for the last section.

- Write a class named `Table` which displays a table of data. Note: you have already done it in Lab05.
- Write a class named `Histogram` which displays a horizontal histogram or a vertical histogram
- Write a class named `IP_address` which calculates and displays statistics information of the source host, such as the frequency count, the sum of packet sizes (i.e. the total number of bytes transferred) and the average of packet sizes.
- Write a class named `A1` which reads the trace file and calculates the statistics information for each source host.
- The `A1GUI` class is the GUI of the main program. The GUI components have been completed for you. You are required to complete the event handling functions.

Stage 1: The Table class - Complete this part in CodeRunner (Question 1 – 3)
Write a class named `Table` which displays a table of data. For example: consider the following code fragment:

```
my_table = Table([['Alice', 24], ['Bob', 19]], headers = ['Name', 'age'], width=10)
my_table.draw_table()
```

The output would be:

```
|Name      age       |
|--------------------|
|Alice     24        |
|Bob       19        |
```

Stage 2: The Histogram class - Complete this part in CodeRunner (Question 4 – 9)
Write a class named `Histogram` which displays a horizontal histogram or a vertical histogram.

Part 1: Horizontal Histogram
For example: consider the following code fragment:

```
h2 = Histogram([['May', 7, 1], ['Bob', 4, 5], ['Mike', 2, 8]], x_width=5, y_width=10)
h2.draw_horizontal_histogram(x_index=0, y_index=1, unit=1)
```

A horizontal histogram is created and the output is:

```
May  |*******
Bob  |****
Mike |**
```

The method takes data at *x_index* to generate the x-label of the histogram and takes data *y_index* to create the histogram. If the scaling `unit` is 1, you will use the numbers to plot the histogram without any scaling. Note: the column widths are set when the object is created.

And consider the following code fragment:

```
h1 = Histogram([['John', 24], ['Bob', 19]])
h1.draw_horizontal_histogram(unit=10)
```

The default values are as follow:

- x_width: 5
- y_width: 10
- x_index: 0
- y_index: 1

Since the scaling `unit` is 10, you will use the numbers at index 1 and divided by 10. (i.e. 24//10, the result is 2. So 2 "*" will be displayed in the first row.) The output is:

```
John |**
Bob  |*
```

Part 2: Vertical Histogram

For example: consider the following code fragment:

```
h2 = Histogram([['May', 3], ['Bob', 7], ['Mike', 2]])
h2.draw_vertical_histogram(unit=1)
```

The width of each column is specified by `x_width`. You may assume that all data can be printed inside each column nicely. Therefore, a vertical histogram is created and the output is:

```
     *
     *
     *
     *
*    *
*    *    *
*    *    *
---------------
May  Bob  Mike
```

Stage 3: The IP_address class - Complete this part in CodeRunner (Question 10 – 16)

Write a class named `IP_address` which calculates and displays statistics information of the source host, such as the frequency count, the sum of packet sizes (i.e. the total number of bytes transferred) and the average of packet sizes. For example: consider the following code fragment:

```
ip_key = '192.168.0.24'
data_list =[(0, 84), (1, 84), (2, 84), (3, 84), (4, 84), (5, 84), (6, 84), (7, 84), (8, 84), (9, 84), (10, 84), (11, 84), (12, 84), (13, 84), (14, 84), (15, 84), (16, 84), (17, 84), (18, 84), (19, 84), (20, 84)]
size = 3
ip = IP_address(ip_key, data_list, size)
```

The `data_list` variable contains a list of tuple objects. The `size` variable determines the number of elements in the three result lists. Consider the above code fragment, the list of tuple objects is divided into 3 groups. The range of the first group is from *0 to 9*, *10-19*, and *20-29*. Statistic information will be calculated in each group:

range	Range: 0 - 9	Range 10-19	Range 20-29
data	(0, 84), (1, 84), (2, 84), (3, 84), (4, 84), (5, 84), (6, 84), (7, 84), (8, 84), (9, 84)	(10, 84), (11, 84), (12, 84), (13, 84), (14, 84), (15, 84), (16, 84), (17, 84), (18, 84), (19, 84)	(20, 84)
frequency	10	10	1
sum	840	840	84
average	84.0	84.0	84.0

Therefore, the `IP_address` object should contain the following:

- IP address: '192.168.0.24'
- The frequency list: [10, 10, 1]
- The sum of packet-size list: [840, 840, 84]
- The average of packet-size list: [84.0, 84.0, 84.0]

You are required to do the above calculation in the constructor of the `IP_address` class and implement the accessor methods to return the corresponding information:

Code fragment:		Output
`print(ip.get_freq_list())`	a list of frequency lists	`[[0, 10], [1, 10], [2, 1]]`
`print(ip.get_sum_list())`	a list of the sum of packet-size lists	`[[0, 840], [1, 840], [2, 84]]`
`print(ip.get_avg_list())`	a list of the average packet-size lists	`[[0, 84.0], [1, 84.0], [2, 84.0]]`
`print(ip.get_statistics())`	the IP address, total sum, total frequency, and average of all packet sizes	`['192.168.0.24', 1764, 21, 84]`

Example2: Consider the following example:

```
ip_key = '192.168.0.2'
data_list = [(33, 60), (34, 64), (34, 1500), (34, 712), (35, 52), (35, 60), (36, 52), (36, 287),
(37, 52), (37, 52), (37, 52), (39, 60), (40, 643), (40, 52)]
size = 5
ip = IP_address(ip_key, data_list, size)
```

You are required to put the list of tuple objects into 5 group:

range	0-9	10-19	20-29	30-39	40-49
				(33, 60), (34, 64), (34, 1500), (34, 712), (35, 52), (35, 60), (36, 52), (36, 287), (37, 52), (37, 52), (37, 52), (39, 60)	(40, 643), (40, 52)
frequency	0	0	0	12	2
sum	0	0	0	3003	695
average	0	0	0	250.2	347.5

The `IP_address` object should contain the following:

Code fragment:	Output
`print(ip.get_freq_list())`	`[[0, 0], [1, 0], [2, 0], [3, 12], [4, 2]]`
`print(ip.get_sum_list())`	`[[0, 0], [1, 0], [2, 0], [3, 3003], [4, 695]]`
`print(ip.get_avg_list())`	`[[0, 0], [1, 0], [2, 0], [3, 250.2], [4, 347.5]]`
`print(ip.get_statistics())`	`['192.168.0.2', 3698, 14, 264]`

You also need to complete the `__str__`, the `__eq__` and the `__lt__` method in the `IP_address` class.

- The `_str__()` method returns a nicely formatted string representation of the object. For example, the method should produce the following output with the FIRST example:
  `192.168.0.24:freq=[10, 10, 1],sum=[840, 840, 84],avg=[84.0, 84.0, 84.0]`
- The `__eq__()` method takes another `IP_address` object and compares if they are equal. Two `IP_address` objects are equal if their ip addresses are the same.
- The `__lt__()` method takes another `IP_address` object. For example, "`192.168.0.2`" is less than "`192.168.0.10`". You should compare the last part of the numbers for ordering.

Stage 4: The A1 class & Helper functions - Complete this part in CodeRunner (Question 17– 23)

Part 1:
Write a class named `A1` which reads the trace file and calculates the statistics information for each source host. For example: consider the following code fragment:

```
my_ip_list = A1('trace33.txt')
print(my_ip_list)
```

The output is:

```
['192.168.0.1:freq=[0, 0, 0, 2],sum=[0, 0, 0, 120],avg=[0, 0, 0, 60.0]', '192.168.0.2:freq=[0,
0, 0, 1],sum=[0, 0, 0, 60],avg=[0, 0, 0, 60.0]', ... '192.168.0.24:freq=[10, 10, 10,
4],sum=[840, 840, 840, 336],avg=[84.0, 84.0, 84.0, 84.0]']
```

You are required to complete some helper functions:

- `read_file()` which accepts a filename and returns a list of tuple objects.
- `get_unique_ip_data_list()` which accepts a list of tuple objects and returns a unique dictionary.

For example, `read_file('trace05.txt')` would return a list of tuple objects:

```
[('192.168.0.24', 0, 84), ('192.168.0.24', 1, 84), ('192.168.0.24', 2, 84), ('192.168.0.24', 3,
84), ('192.168.0.24', 4, 84)]
```

Then, `get_unique_ip_data_list()` function takes the above list of tuple objects and returns a dictionary:

```
{'192.168.0.24': [(0, 84), (1, 84), (2, 84), (3, 84), (4, 84)]}
```

Use the above two functions to read data from the parameter text file and create a list of `IP_address` objects. Implement the constructor and the `__str__` method to print the content in the list of `IP_address` objects.

Part 2:
You are required to implement 4 methods. The `get_statistics()` method returns a list of statistics information from the list of IP addresses. The statistics information includes the IP address, the total packet sizes, the frequency count and the average in each `IP_address` object. For example, consider the following code fragment:

```
my_ip_list = A1('trace33.txt')
print(my_ip_list.get_statistics())
```

The output is:

```
[['192.168.0.1', 120, 2, 60], ['192.168.0.2', 60, 1, 60], ['192.168.0.8', 60, 1, 60],
['192.168.0.10', 120, 2, 60], ['192.168.0.11', 60, 1, 60], ['192.168.0.12', 60, 1, 60],
['192.168.0.15', 2336, 4, 584], ['192.168.0.17', 60, 1, 60], ['192.168.0.24', 2856, 34, 84]]
```

The `get_freq_table`() takes an IP address as a parameter and returns a list of frequency counts of the IP address.
The `get_sum_table`() takes an IP address as a parameter and returns a list of the sum of packet sizes of the IP address.
The `get_avg_table`() takes an IP address as a parameter and returns a list of the average of the packet sizes of the IP address.

Code fragment:	Output
`print(my_ip_list.get_freq_table('192.168.0.15'))`	`[[0, 0], [1, 0], [2, 0], [3, 4]]`
`print(my_ip_list.get_sum_table('192.168.0.24'))`	`[[0, 840], [1, 840], [2, 84]]`
`print(my_ip_list.get_avg_table('192.168.0.24'))`	`[[0, 84.0], [1, 84.0], [2, 84.0]]`

Stage 5: The complete program - Complete this part in CodeRunner (Question 24)
You are required to combine all classes together. Consider the following code fragment:

```
my_ip_list = A1('trace33.txt')
my_result = my_ip_list.get_statistics()
t1 = Table(my_result, headers=['IP address', 'Sum', 'Freq', 'Avg'], width=13)
t1.draw_table()
```

The output would be:

```
|IP address   Sum          Freq         Avg          |
|-------------------------------------------------|
|192.168.0.1  120          2            60           |
|192.168.0.2  60           1            60           |
|192.168.0.8  60           1            60           |
|192.168.0.10 120          2            60           |
|192.168.0.11 60           1            60           |
|192.168.0.12 60           1            60           |
|192.168.0.15 2336         4            584          |
|192.168.0.17 60           1            60           |
|192.168.0.24 2856         34           84           |
```

And consider the following code fragment:

```
my_result = my_ip_list.get_freq_table('192.168.0.24')
h1 = Histogram(my_result, x_width=3)
```

The output would be:

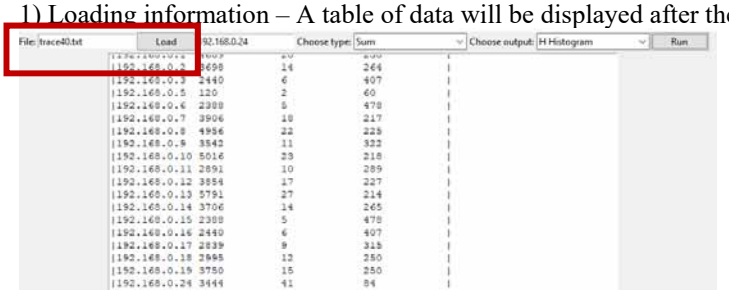
```
0  |**********
1  |**********
2  |**********
3  |****
```

```
*  *  *  *
*  *  *  *
*  *  *  *
*  *  *  *
*  *  *  *
*  *  *  *
*  *  *  *
*  *  *  *
------------
0  1  2  3
```

Stage 6: A1GUI – The complete program with GUI

Complete two functions to handle the button click events and display the corresponding statistics information. You may need to do some extra reading for this section.

1) Loading information – A table of data will be displayed after the "Load" button is clicked.



2) Create Table/Horizontal Histogram/Vertical Histogram:
Users will be able to choose the type of statistics information (i.e. frequency, sum or average) and the display type (i.e. table, horizontal histogram or vertical histogram) to display the statistics data of the specified source host. For example:

Source host: `192.168.0.24`; Type: `Sum`; Output type: `Horizontal histogram`

```
Sum
0  |********
1  |********
2  |********
3  |********
4  |
```

Source host: `192.168.0.24`; Type: `Frequency`; Output type: `Table`

```
Freq
|Time Freq |
|----------|
|0    10   |
|1    10   |
|2    10   |
|3    10   |
|4    1    |
```

Source host: `192.168.0.24`; Type: `Average`; Output type: `Vertical histogram`

```
Average
*  *  *  *  *
*  *  *  *  *
*  *  *  *  *
*  *  *  *  *
*  *  *  *  *
*  *  *  *  *
*  *  *  *  *
*  *  *  *  *
---------------
0  1  2  3  4
```

Submission

Submit your assignment online via the assignment dropbox (https://adb.auckland.ac.nz/) at any time from the first submission date up until the final date. You will receive an electronic receipt. Submit **ONE A1.zip** file containing all source files (i.e. new, changed, and unchanged) – remember to include your name, UPI and a comment at the beginning of each file you create or modify. You may make more than one submission, but note that every submission that you make replaces your previous submission. Submit **ALL** your files in every submission. Only your very latest submission will be marked. Please double check that you have included all the files required to run your program and A1.txt (see below) in the zip file before you submit it**. Your program must compile and run to gain any marks. We recommend that you check this on the lab machines before you submit.**

ACADEMIC INTEGRITY

The purpose of this assignment is to help you develop a working understanding of some of the concepts you are taught in the lectures. We expect that you will want to use this opportunity to be able to answer the corresponding questions in the tests and exam. We expect that the work done on this assignment will be your own work. We expect that you will think carefully about any problems you come across, and try to solve them yourself before you ask anyone for help. The following sources of help are acceptable:

- Lecture notes, tutorial notes, skeleton code, and advice given by us in person or online, with the exception of sample solutions from past semesters.
- The textbook.
- The official Java documentation and other online sources, as long as these describe the general use of the methods and techniques required, and do not describe solutions specifically for this assignment.
- Piazza posts by, or endorsed by an instructor.
- Fellow students pointing out the cause of errors in your code, without providing an explicit solution.

The following sources of help are not acceptable:

- Getting another student, friend, or other third party to instruct you on how to design classes or have them write code for you.
- Taking or obtaining an electronic copy of someone else's work, or part thereof.
- Give a copy of your work, or part thereof, to someone else.
- Using code from past sample solutions or from online sources dedicated to this assignment.

The Computer Science department uses copy detection tools on all submissions. Submissions found to share code with those of other people will be detected and disciplinary action will be taken. To ensure that you are not unfairly accused of cheating:

- Always do individual assignments by yourself.
- Never give any other person your code or sample solutions in your possession.
- Never put your code in a public place (e.g., Piazza, forum, your web site).
- Never leave your computer unattended. You are responsible for the security of your account.
- Ensure you always remove your USB flash drive from the computer before you log off, and keep it safe.

Marking Scheme

CodeRunner: Correctness

Stage 1: `Table`	12	Coderunner: No errors, program always works correctly and meets the specification(s).
Stage 2: `Histogram`	18	Coderunner: No errors, program always works correctly and meets the specification(s).
Stage 3: `IP_address`	21	Coderunner: No errors, program always works correctly and meets the specification(s).
Stage 4: `A1`	21	Coderunner: No errors, program always works correctly and meets the specification(s).
Stage 5:Program	4	Coderunner: No errors, program always works correctly and meets the specification(s).

A1:

Stage 6: GUI	4	The load_file function is implemented correctly.
Stage 6: GUI	6	Result is displayed correctly.
Others:	2	Your program contains a docstring with your username and a description of your program.
	4	Code is well-commented
	1	The `Table` class: No errors, code is clean, understandable, and well-organized
	1	The `Table` class: Code always uses the best approach
	1	The `Histogram` class: No errors, code is clean, understandable, and well-organized.
	1	The `Histogram` class: Code always uses the best approach
	1	The `IP_address` class: No errors, code is clean, understandable, and well-organized.
	1	The `IP_address` class: Code always uses the best approach
	1	The `A1` class: No errors, code is clean, understandable, and well-organized.
	1	The `A1` class: Code always uses the best approach