BASH SHELL – FILE UTILITIES / AWK: RECORD & FIELD MANAGEMENT

For this handout, we will consider usage data from Citi Bike – a bike share organization in NYC.



<https://www.citibikenyc.com/>

The goal of our work will be to understand the rental / usage patterns for the following holidays:

New Years Day, Easter, Memorial Day, Independence Day, Labor Day, Thanksgiving, and Christmas

The 2016 usage rental data will be used for our investigation. The following table provides the actual dates for the holidays listed above for 2016.

|  |  |
| --- | --- |
| Holiday | Date |
| New Years Day | January 1, 2016 |
| Easter | March 27, 2106 |
| Memorial Day | May 30, 2016 |
| Independence Day | July 4, 2016 |
| Labor Day | September 5, 2016 |
| Thanksgiving | November 24, 2016 |
| Christmas | December 25, 2016 |

The rental / usage data is available from Citi Bike’s System Data website.

<https://s3.amazonaws.com/tripdata/index.html>



AUTOMATING FILE DOWNLOAD

The typical method of retreiving data from the internet is to use a web browser. A web browser has many advantages in getting data. However, if many files need to be retrieved, then automating the process of downloading files should be considered.

The curl command in BASH can be used to download. The –O option writes the output to a local file using the same name as the file being downloaded. Curl is a BASH tool to transfer data from a server to your machine or vise versa. Curl supports a variety of transfer protocols, e.g. FTP, HTTP, HTTPS, TELNET, etc.

$ curl -O https://s3.amazonaws.com/tripdata/201601-citibike-tripdata.zip

The system data is provided by month; thus, in addition to January we will need to get data from March, May, July, September, November, and December. The following commands can be used to retrieve the data from these remaining months.

$ curl -O https://s3.amazonaws.com/tripdata/201603-citibike-tripdata.zip

$ curl -O https://s3.amazonaws.com/tripdata/201605-citibike-tripdata.zip

$ curl -O https://s3.amazonaws.com/tripdata/201607-citibike-tripdata.zip

$ curl -O https://s3.amazonaws.com/tripdata/201609-citibike-tripdata.zip

$ curl -O https://s3.amazonaws.com/tripdata/201611-citibike-tripdata.zip

$ curl -O https://s3.amazonaws.com/tripdata/201612-citibike-tripdata.zip

Comment: The following variation of the curl command can be used to download all months.

$ curl -O https://s3.amazonaws.com/tripdata/20160[1-9]-citibike-tripdata.zip

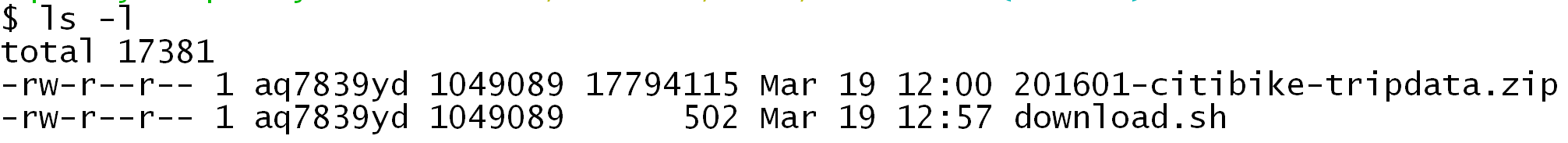
$ curl -O https://s3.amazonaws.com/tripdata/20161[0-2]-citibike-tripdata.zip

Another alternative to running each of these commands one-at-a-time, would be to place all necessary commands into a single text file. Here the curl command for each month has been put into a file called download.sh. The \*.sh extension just identifes that the file contains Bash shell commands.



The download.sh file is put in the same directory for which the files are to be downloaded.

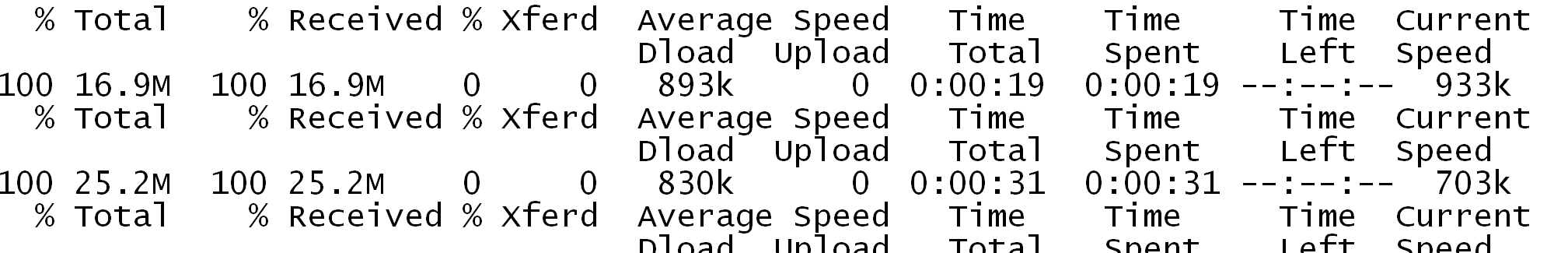
$ ls –l



The following commans will exectue all the commands found in the download.sh file.

$ bash download.sh

The curl commands provides infromation about the sequential downloading of the requested files. A snip-it of the download statistics regarding the first couple files are provided here.



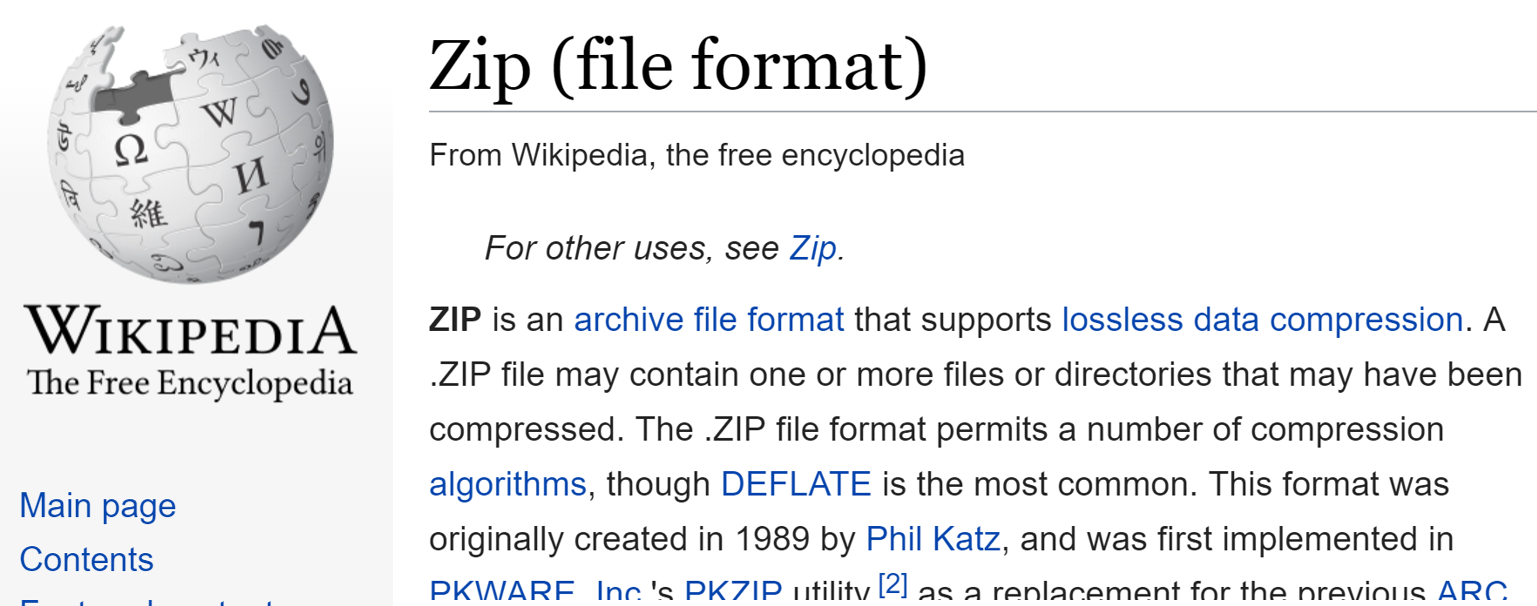
Checking to make sure all files have been downloaded correctly.

$ ls –l



UNZIPPING FILES

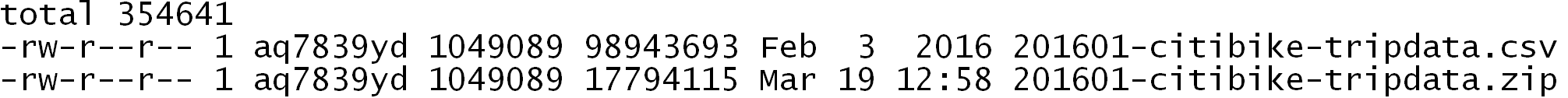
The files provided by Citibike are in a zip format. Zip is a common data compresssion format which reduces the size of a file. Before one can use a zip file, the content of the file must be extracted or unzipped. There are several programs available to unzip a file. Bash has a built in unzip command.



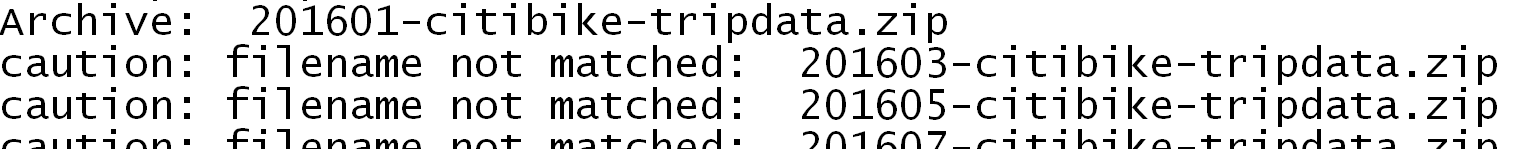
The following command can be used to unzip a zipped file in Bash. Notice that the size of the compressed file is about 20% (17794115 / 98943693) of the size of the original data file.

$ unzip 201601-citibike-tripdata.zip

$ ls –l



Unfortantely, the wildcard character is not understood by unzip in the following usage. However, if the \*.zip is placed in a quoted string, e.g. ‘\*.zip’, then the unzip function interprets the wildcard correctly.

$ unzip \*.zip 

FOR LOOPS

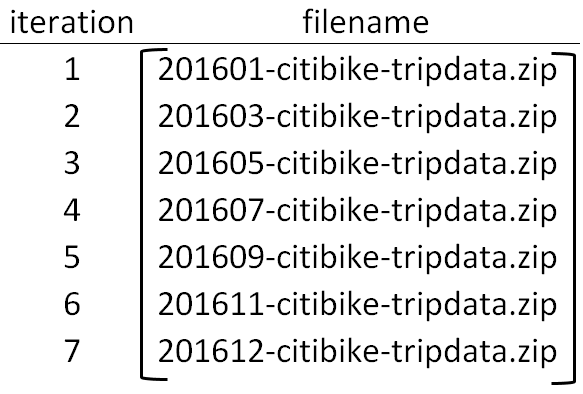
A for loop can be used to systematically apply a set of commands. The simple example below prints the values 1 through 5 to the screen.

|  |  |
| --- | --- |
| Generic structure of a for loop in Bash | Simple example in Bash |
|  | $ for i in {1..5}  > do  > echo $i  > done  This for loop simple prints the values 1, 2, 3, 4, 5. The $i is used to refer to the value of i through each iteration through the loop. This loop starts with i=1 and stops at i=5. |

In our application, we need to systematically unzip the \*.zip files in our directory. The variable in this loop is filename and the wildcard character automatically creates the appropriate variable\_range for the filename variable. Use the following commands to unzip each fo the zip files in the current directory.

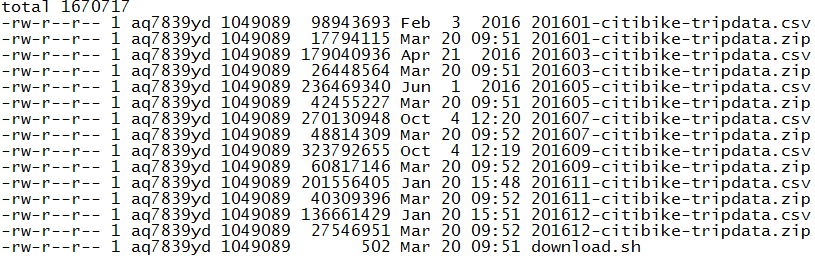
$ for filename in 2016\*-citibike-tripdata.zip  
> do  
> unzip $filename  
> done

A schemtic showing the filename for each iteration of the above loop.



A quick glance to ensure all files have been successfully unzipped.

$ ls –l

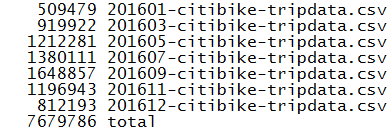


EXTRACTING RECORDS / ROWS

The goal in our investigation is to consider rental / usage patterns of bikes for the following seven holidays. The data needed will need to be filtered out of these data files. This filtering cannot be done in Excel as the number of records in these datafiles exceeds Excel’s capabiliites.

|  |  |
| --- | --- |
| Holiday | Date |
| New Year’s Day | January 1, 2016 |
| Easter | March 27, 2106 |
| Memorial Day | May 30, 2016 |
| Independence Day | July 4, 2016 |
| Labor Day | September 5, 2016 |
| Thanksgiving | November 24, 2016 |
| Christmas | December 25, 2016 |

The following command provides a line count for each of the \*.csv files. The number of records in these files exceed what Excel can handle.

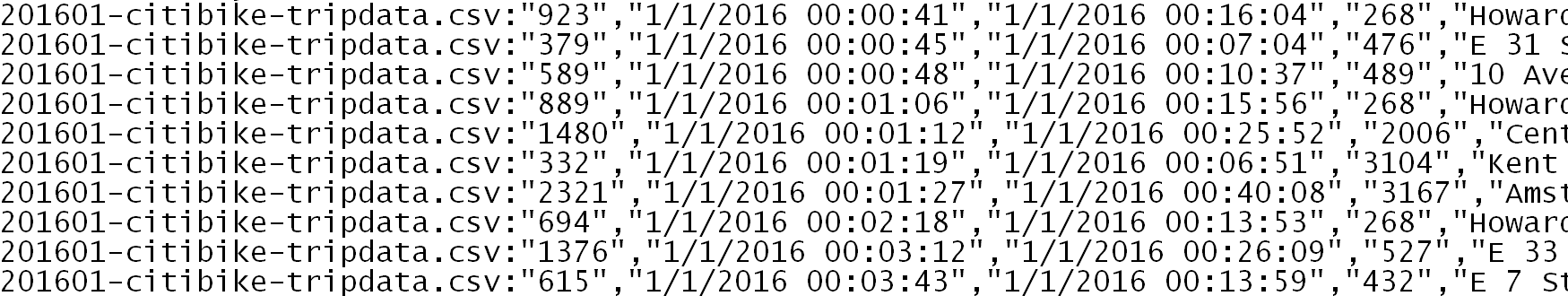
$ wc –l \*.csv  


The next task is to filter the necessary rows from these files. This can be accomplished with the following grep command. The output is being saved into a file called Holidays.csv.

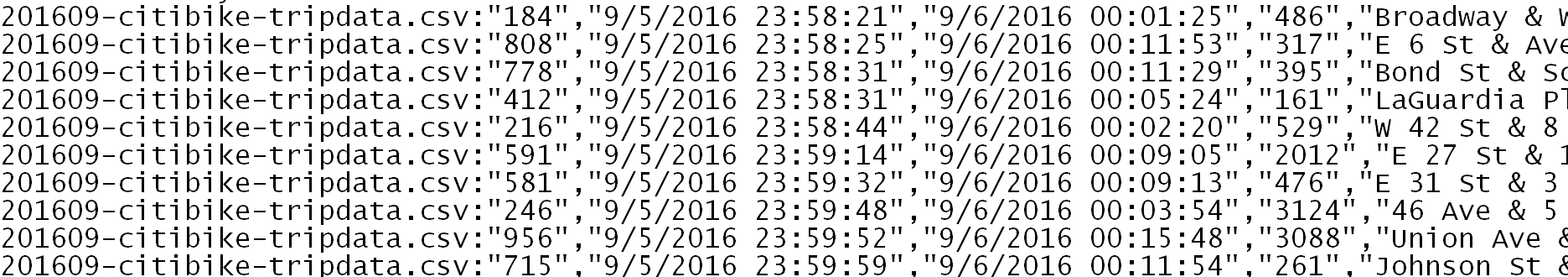
$ grep '1/1/2016\|3/27/2016\|5/30/2016\|7/4/2016\|9/5/2016\|11/24/2016\|12/25/2016' 2016\*-citibike-tripdata.csv > Holidays.csv

Looking at the first few lines and the last few lines of the Holidays.csv file.

$ head Holidays.csv

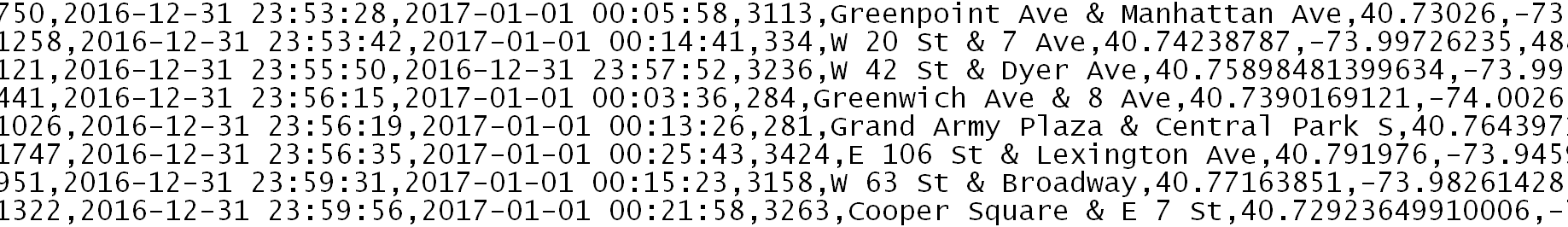


$ tail Holidays.csv



The Holidays.csv file does not contain bike rentals beyond Labor Day. The reason for this is that the date format is different for the 201611-citibike-tripdata.csv and 201612-citibike-tripdata.csv files.

$ tail 201612-citibike-tripdata.csv



The following table provides the date structure for each Holiday in these data files. For some reason, the system data changed their data format between September 2016 and November 2016.

|  |  |
| --- | --- |
| Holiday | Date |
| New Years Day | 1/1/2016 |
| Easter | 3/27/2106 |
| Memorial Day | 5/30/2016 |
| Independence Day | 7/4/2016 |
| Labor Day | 9/5/2016 |
| Thanksgiving | 2016-11-24 |
| Christmas | 2016-12-25 |

Given this information, modifications will need to be made to grep command.

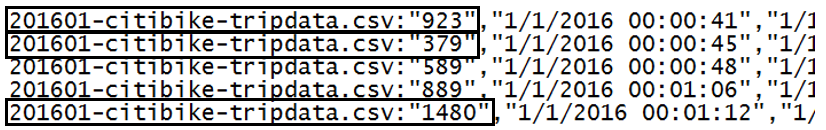
$ grep '1/1/2016\|3/27/2016\|5/30/2016\|7/4/2016\|9/5/2016\|2016-11-24\|2016-12-25' 2016\*-citibike-tripdata.csv > Holidays.csv

A quick check to ensure that bike rentals for Thanksgiving and Christmas are included.

$ tail Holidays.csv



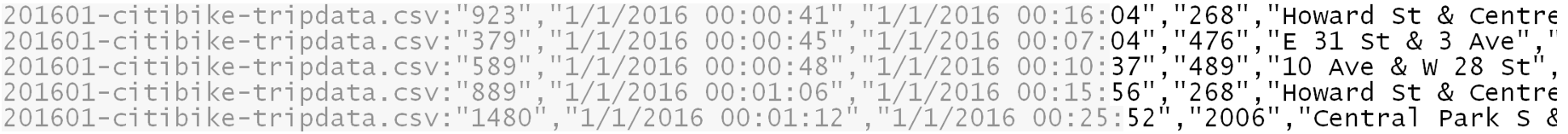
Notice that the grep command has contamimated the first field in this data file. In particiular, the frist field is the amount of time (in seconds) for the bike rental. After running the grep command this field also contains the filename from which this line was obtained.



Consider the following command as an attempt to remove the filename information in the first field. The head command here is being used to see the first 10 lines of the output file – the output is not yet being written out to a file.

$ sed -n 's/^.\*://p' Holidays.csv | head -10

The command above prints the content of each line after the *last* colon. Keeping the content of each line after the first colon is needed here.



Consider the following alternatives to the above command that appear to remove the filename content from the 1st field.

Command #1: $ sed -n 's/^.\*csv://p' Holidays.csv | head -10

Command #2: $ sed -r 's/.{29}//' Holidays.csv | head -10

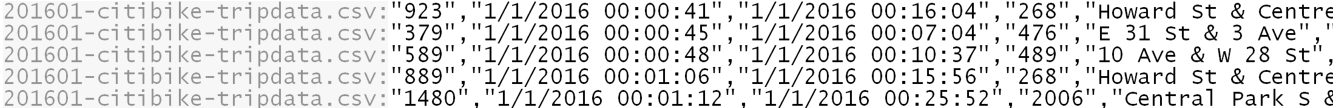
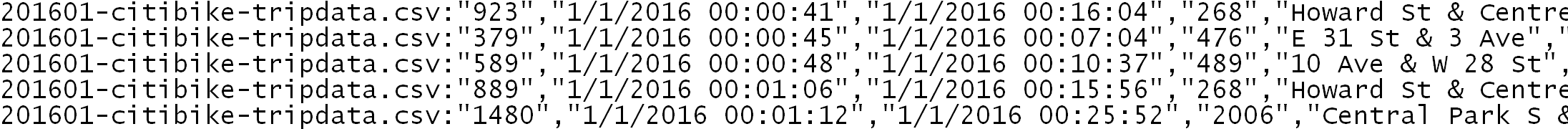
Command #3: $ sed 's/^[^:]\*://' Holidays.csv | head -10

Questions

1. What is the substitute command finding in Command #1? What is the substitue command replacing in Command #1? What would happen if the sed command were changed to ‘s/.csv://p’? Verify that Command #1 produces the desired output.
2. What is the purpose of .{29} in Command #2? Verify that Command #2 produces the desired output.
3. In Command #3, the first ^ is for beginning of line, the [^:] is syntex for not a colon, the \*: tells sed to grab any number of characters until you find a colon. Does Command #3 work?

The following command will remove the filename information from the first field. In-place editing is being used, thus the output will be placed back into the Holidays.csv file.

$ sed –i 's/^[^:]\*://' Holidays.csv

The last step will be to add back in the header information. This can be done using sed (Command #1) or one could grab the first line from one of the original files and then append it to the Holidays.csv file (Command #2). The output is being saved into a final called Holidays\_Final.csv

Command #1:

$ sed '1 i\tripduration, starttime, stoptime, start station id, start station name, start station latitude, start station longitude, end station id, end station name, end station latitude, end station longitude, bikeid, usertype, birth year, gender' Holidays.csv > Holidays\_Final.csv

Command #2:

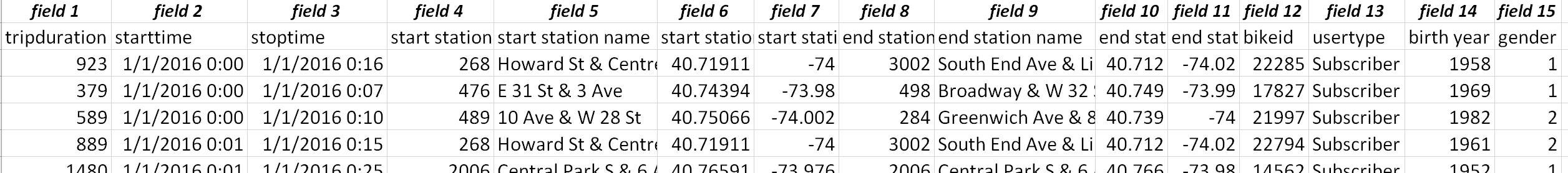
$ head -1 201601-citibike-tripdata.csv > Holidays\_Header.csv

$ cat Holidays\_Header.csv Holidays\_v2.csv > Holidays\_Final.csv

EXTRACTING FIELDS / COLUMNS

Sed and grep can be used to filter data, i.e. extract rows from a data file. The task of extracting fields (or columns) from data files can be accomplished through commands such as cut or awk.

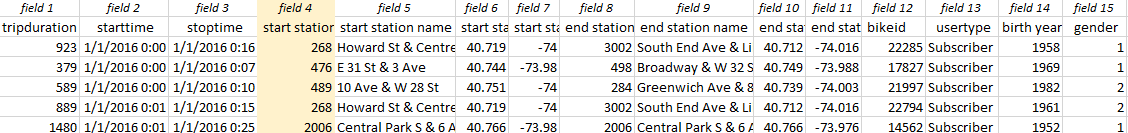
The following schmatic shows the fields / columns for the Citibike data.



CUT UTILITY

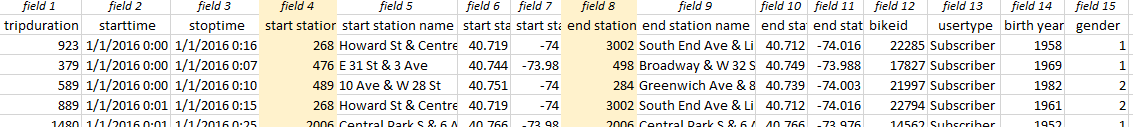
The following use of the cut command will extract or cut out field 4, i.e. statition id, from the dataset. The –d option is specifying the , as the delimited characters – this should be encapulasted with a backlash as show.

$ cut -f4 -d\, Holidays\_Final.csv



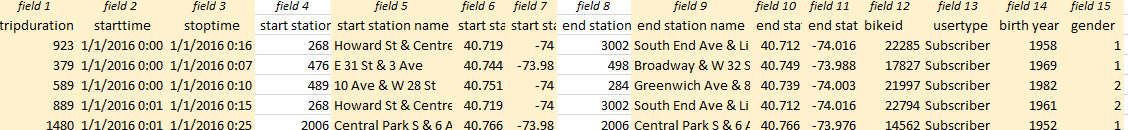
The following can be used to extract field 4 (start station id) and field 8 (end station id) from the dataset.

$ cut -f4,8 -d\, Holidays\_Final.csv



The use of the --complment option will extract all fields except field 4 and field 8.

$ cut --complement -f4,8 -d\, Holidays\_Final.csv



One nice feature regarding the cut utility is that it is easy to use. However, cut does have some limitation. For example, cut does not have the ability to evaluate the value contained with a field. However, AWK is a data-driven scripting language does have this ability.

Awk can be used to manage fields (and records) in a data file. According to Wikipedia, AWK was initially developed in 1977 by Alfred Aho, Peter Weinberger, and Brian Kernighan. AWK is more powerful than cut. Some applications of awk are provided below.

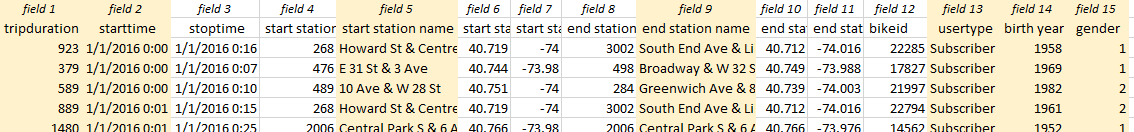


The following awk command prints fields 1, 2, 5, 9, 13, 14, and 15. A description of the options is provided here:

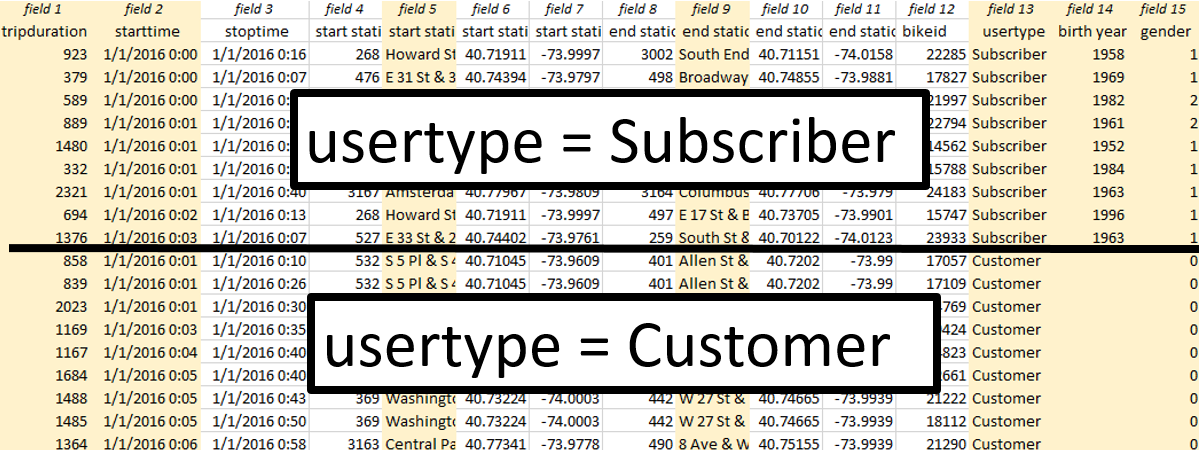
* The –F option specifies that the delimiter in the infile is a comma.
* The –v OFS=”,” specifies that the field separator for the outfile should be a comma.
* Fields within the file are generically labelled $1, $2, etc. Thus, {print $1, $2, $5, $9, $13, $14, $15} will print fields 1, 2, 5, 9, 13, 14, and 15 to the output file.

$ awk -F, -v OFS=, '{print $1,$2,$5,$9,$13,$14,$15}' HOlidays\_Final.csv > Holidays\_LessFields.csv

The command above will extract fields 1, 2, 5, 9, 13, 14, and 15.



Awk can be used to create two mutually exclusive datasets - one for usertype = Subscriber and another for usertype = Customer. Subscribers are those that pay a month fee for the bike share program and customers are those that use the bike share program, but are not monthly subscribers, e.g. a tourist would likely be a customer and not a subscriber.



The following table provides counts for this data that can be used to verify the number of records in the following awk commands.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Usertype  *(field 13)* | Gender *(field 15)* | | | Total |
| Unknown (0) | Male (1) | Female (2) |
| Customer | 46129 | 662 | 563 | 47354 |
| Subscriber | 2652 | 85544 | 33313 | 121509 |
| blank | 4 | 366 | 167 | 537 |
| Total | 48785 | 86572 | 34043 | 169400 |

As mentioned above, awk is more powerful than cut. For example, awk has the ability to evaluate a value for a specified field. An action can take place after an evaluation of this value is completed. For example, the following command prints fields 1, 2, 5, 9, 13, 14, 15 only for usertype = Customer. Usertype is field 13; thus $13 is used to refer to the usertype in awk. The tilde character ( ~ ) is used to determine if two values are equal.

Filter: usertype = “Customer”

$ awk -F, -v OFS=, '$13~"Customer" {print $1,$2,$5,$9,$13,$14,$15}' Holidays\_Final.csv > Holidays\_Customers.csv

Doing the same for the Subscriber rows.

Filter: usertype = “Subscriber”

$ awk -F, -v OFS=, '$13~"Subscriber" {print $1,$2,$5,$9,$13,$14,$15}' HOlidays\_Final.csv > Holidays\_Subscribers.csv

Getting the number of lines in each of these files to ensure that the data has been appropriately divided into Customers and Subscribers.

$ wc -l Holidays\_Customers.csv Holidays\_Subscribers.csv

47354 Holidays\_Customers.csv

121509 Holidays\_Subscribers.csv

168863 total

Comment: The header row is not included in these files. The reason is that the header contains the name of the fields and thus most likely does not meet the condition being checked. Awk does appear to have the ability to skip over certain rows. For example, if NR > 1 is used, then the 1st row is skipped; however, the syntax in using this command is beyond the scope presented here.

The following awk command is used to reduce the rows to those that are customers and male. Notice the && syntax is for AND. The syntax for OR is ||.

Filter: usertype = “Customer”

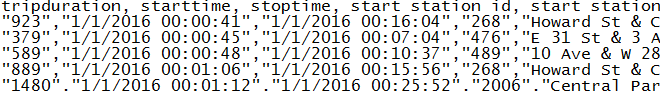
gender = “Male”

$ awk -F, -v OFS=, '$13~"Customer" && $15~”1” {print $1,$2,$5,$9,$13,$14,$15}' HOlidays\_Final.csv > Holidays\_Customers\_Males.csv

Awk has the ability to work with numeric fields as well. For example, suppose the goal is to retain any bike rentals whose trip duration is less than 10 minutes. Note: Tripduration is reported in seconds in this file (10 minutes x 60 seconds/minute = 600 seconds).

Filter: tripduration < 600

The tripduration field is a quoted string in the Holidays\_Final.csv dataset. The quotes must be removed so that Awk can evaluate each value against 600.



The following sed commands can be used to remove the quotes from the first field, i.e. tripduration. Command #1 remove the 2nd quote on each line and Command #2 remove the 1st quote from each line.

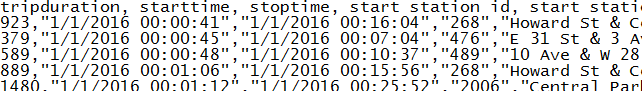
Command #1: $ sed 's/\"//2' Holidays\_Final.csv

Command #2: $ sed 's/\"//' Holidays\_Final.csv

Questions

1. Would it work to skip Command #1 and just run Command #2 twice? Discuss.
2. What would happen if Command #2 were run first, then Command #1 was run? Discuss.

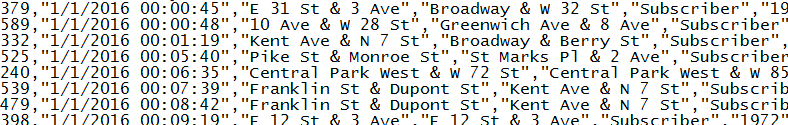
A quick glance of the file after removing the quotes from the first field.



The following awk command will keep lines whose tripduration, i.e. 1st field, is less than 600.

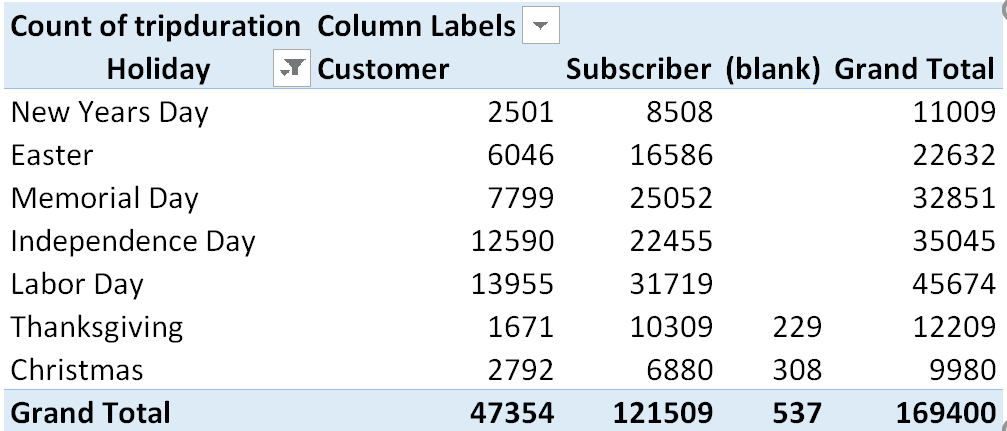
$ awk -F, -v OFS=, '$1 < 600 {print $1,$2,$5,$9,$13,$14,$15}' Holidays\_Final.csv > Holidays\_Lessthan10mins.csv

The tripduration < 600 filter appear to have worked. Notice all values in the 1st field are less than 600.



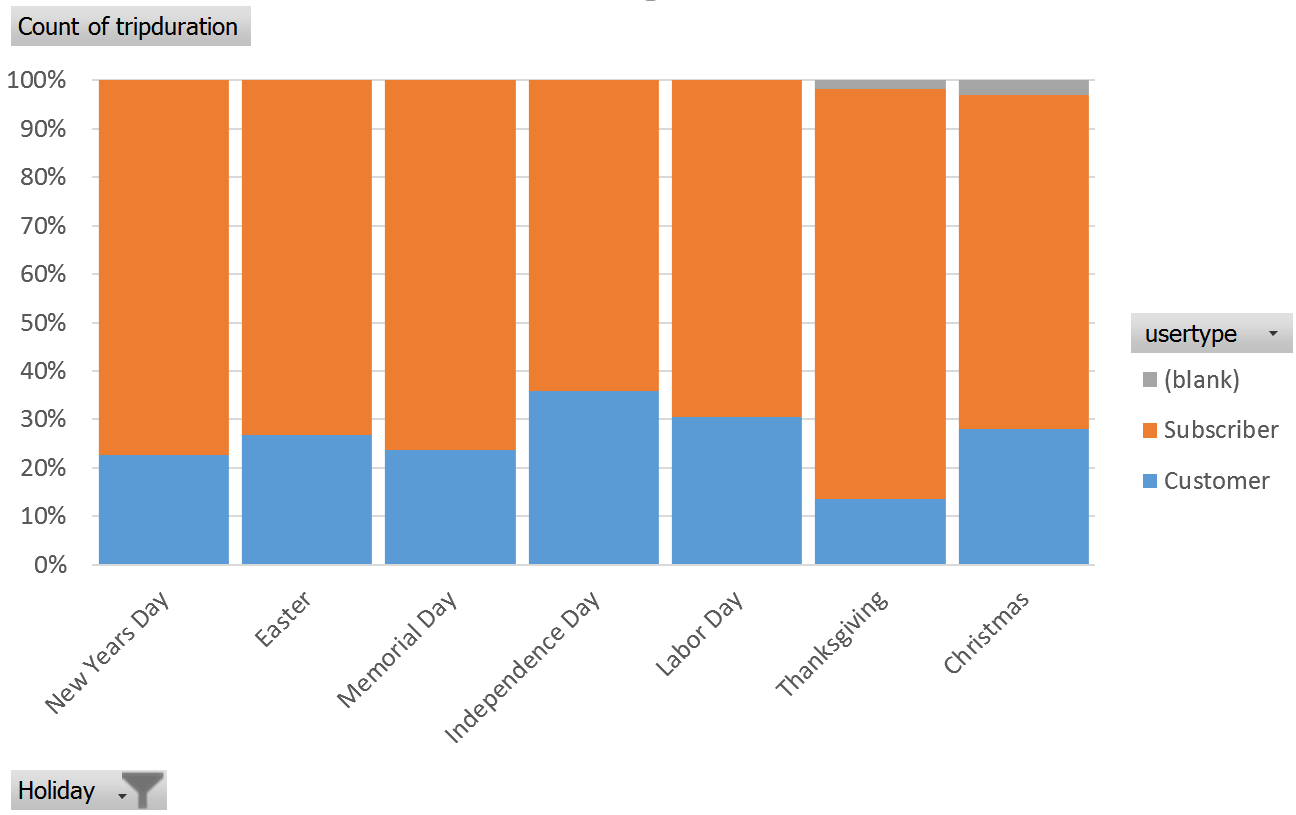
SUMMARIES / VISUALIZATIONS

Much of the summaries here are center on the similaries and differences between the Customers and Subscribers. The following table shows the number of bike rentals for each usertype across the seven holidays.

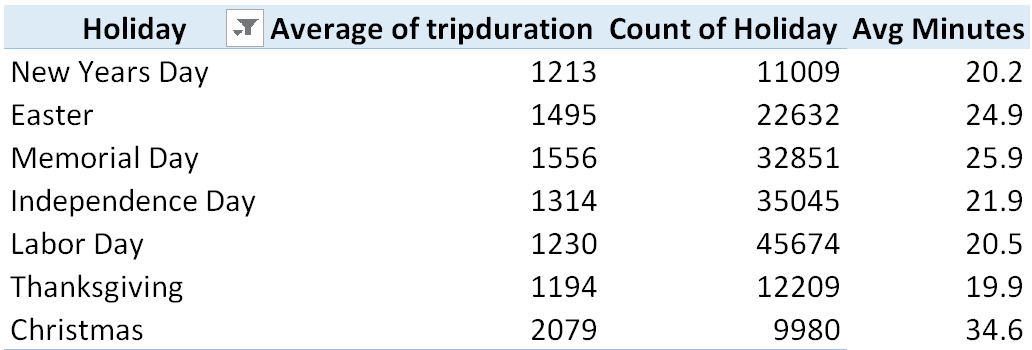


Questions

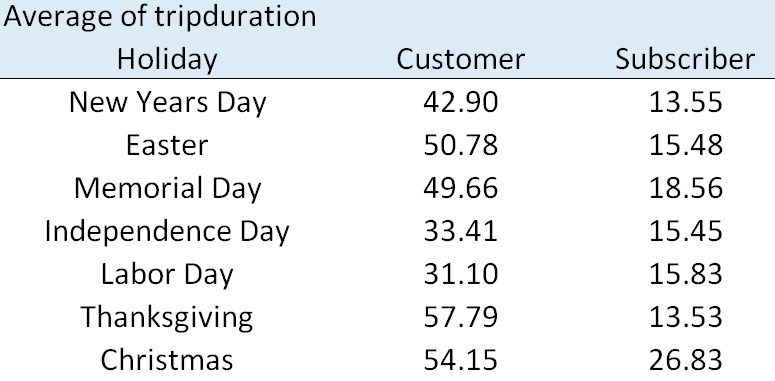
1. What Holidays tend to have the most bike rentals? Is this what you’d expect? Discuss.
2. Consider the following graph shows a breakdown of usertype by each holiday. What information about bike rental patterns is learned from this graph? Discuss.

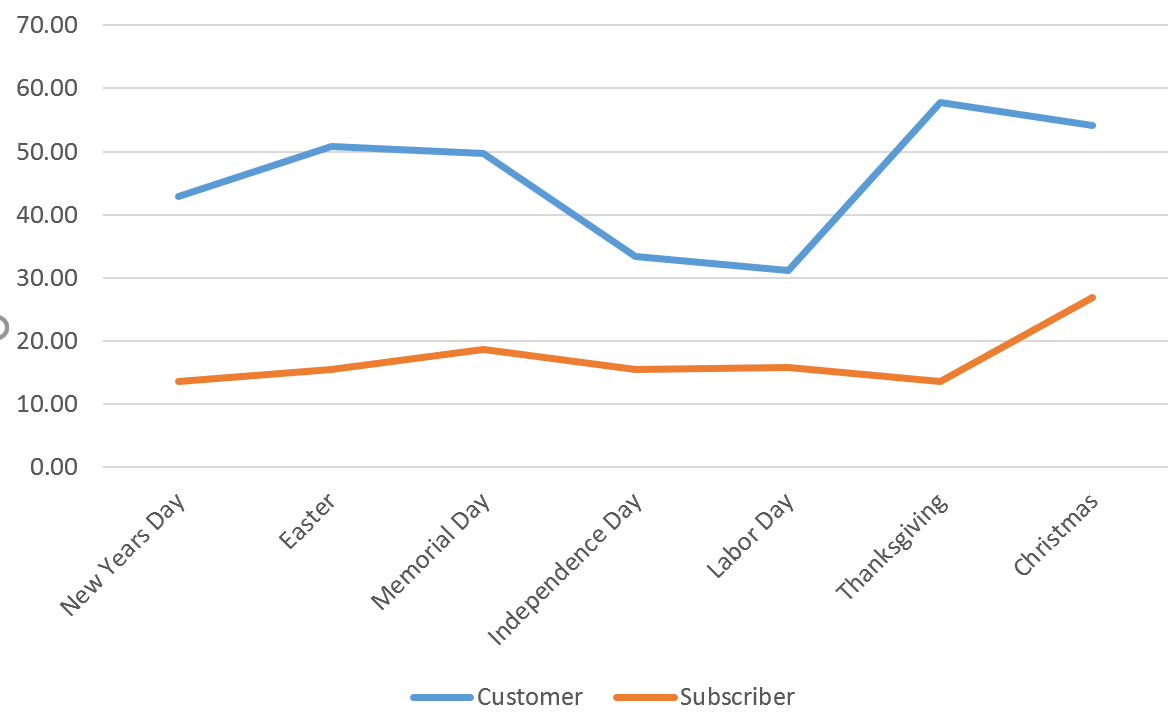


1. The rightmost column in the table below provides the average length of rental for each Holiday. Which Holiday tends to have the highest average trip duration?

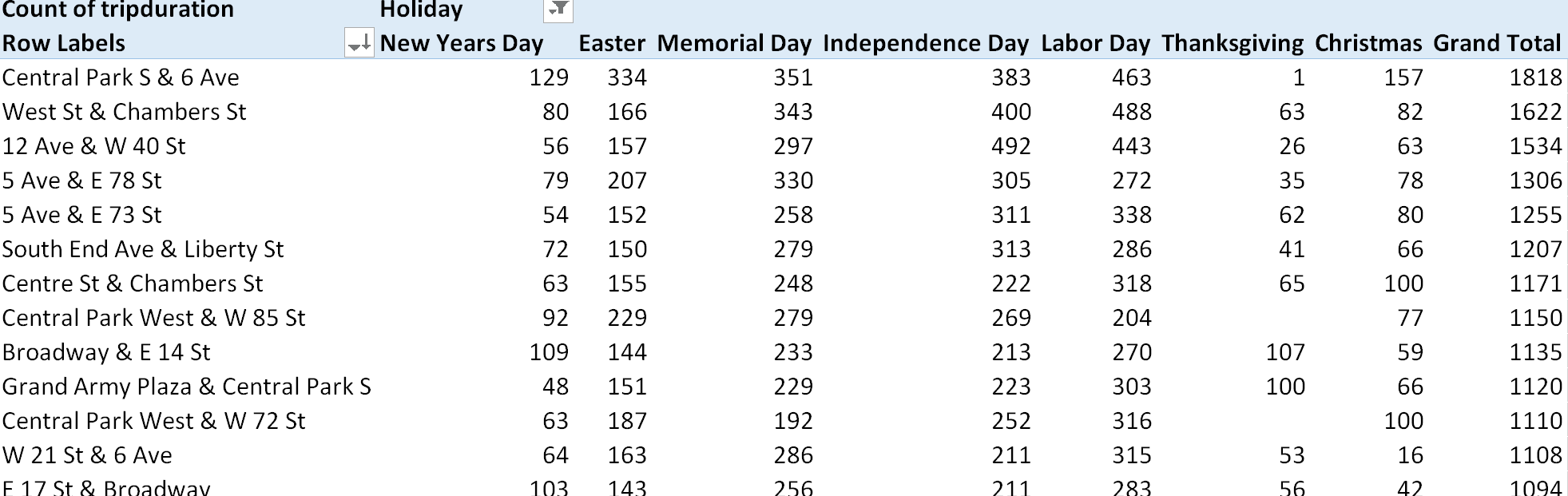


1. The following table and graph provides a breakdown of average trip duration by usertype. Tourist tend to be usertype = Customer, what can be said about tourist trip duration on Holidays? How does their trip duration pattern different from Customers? Discuss.





1. The following table provides a list of the most frequently used bike stations from which rentals take place. Notice that Central Park is listed often. Does the popularity of a bike station depend on the Holiday? Why might this be a problem for those that are managing the bike rental system? Discuss.



Tasks:

Consider the following situation of password creation and hacking. Suppose your name is ANDY and you have decided to use these four letters to create a password that is 8 characters.



Simple probabilities can be computed for various situations. For example, the probability of ANDY being randomly generated on placed into the first four characters is simply (1/4)\*(1/4)\*(1/4)\*(1/4) = 1/256 ≈ 0.39%.

|  |  |
| --- | --- |
| Getting ANDY spelled out correctly | Probability of getting ANDY starting in position 1:  (1/4)\*(1/4)\*(1/4)\*(1/4) = 0.0039 = 1/256 |
| Probability of getting ANDY across all positions = 5 \* (1/256) = 5/256 = 0.0195 ≈ 2% | |

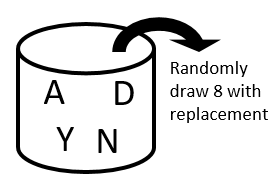
Consider the following commands.

Command #1: $ cat /dev/urandom | tr -dc 'ANDY' | fold -w 8 | head -1000000 > random.txt

Command #2: $ sed 's/.//5g' random.txt | grep 'ANDY' | wc -l

Command #1 does the following:

* Generate a string that is 8 characters long.
* head -1000000 tells bash to generate 1000000 rows
* tr –dc ‘ANDY’ specifies that A, N, D, and Y are the characters available for random selection, i.e. these are the characters in the “bucket” from which letters are drawn randomly



* Output is being saved into a file called random.txt

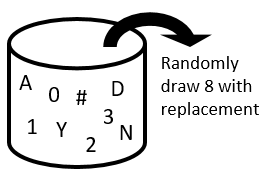
Command #2 does the following:

* Sed command deletes all characters beyond the first four, this output is piped forward to grep
* Grep command searches for string ANDY. If ANDY is present in this line, the line is pushed forward
* The wc -l simply counts the number of lines, i.e. the number of times ANDY is found in the first four characters.

1. Run command #1 and command #2 in Bash. How many times did ANDY show up in the first four positions? Compute the empirical probability, i.e. # times / 1000000. Is the empirical probability close to the theoretical value of 1/256?
2. Modify command #2 so that searching of ANDY is done across all positions (not just the first four positions). How many times did grep find ANDY? Once again, is the empirical probability for this situation close to 5/256?
3. Next, suppose the “bucket” from which the random selection is done contains A, N, D, Y, #, 0, 1, 2, and 3.
4. Consider Command #3 below.

Command #3: $ cat /dev/urandom | tr -dc 'ANDY#[0-3]' | fold -w 8 | head -1000000 > random.txt

This command works but is \*not\* what is needed here. Fix this command so that is produces random passwords as requested.



1. What is the theoretical probability of getting ANDY#1 in a sequence of 8 characters?
2. Use grep and wc –l to compute the empirical probability for this situation. Does the empirical probability agree with theoretical value? Note: You may need to generate more than 1000000 random passwords to verify this probability.
3. Consider the following commands.

* Command #4 generates 10,000,000 random passwords. Each password is 8 characters long and is made up of the characters A, N, D, and Y.
* Command #5 uses sed to search for ANDY in each password. The date; command is placed immediately before and after the sed command. This can be used to compute the length of time it takes sed to do its searching.
* Command #6 uses grep to search for ANDY in each password. Again, the length of time for processing can be computed with date; placed before and after the grep command.

Command #4: $ cat /dev/urandom | tr -dc 'ANDY' | fold -w 8 | head -10000000 > random.txt

Command #5: $ date; sed -n '/ANDY/p' random.txt | wc -l; date;

Command #6: $ date; grep 'ANDY' random.txt | wc -l; date;

1. Is sed or grep faster in doing its searching? Discuss.
2. What happens to the time it takes for sed to complete its searching when the number of random passwords generated is doubled? Does the processing time double?
3. What happens to the time it takes for sed to complete its searching when the length of the password is changed from 8 characters to 16 characters. Does the processing time double?
4. What happens to the time it takes for grep to complete its searching when the length of the password is changed from 8 characters to 16 characters. Does the processing time double?