Homework 3

Name: [Zongzhe Lin]

Collaborators: [Anran Zhao]

Due date: May 19, 2024

Submission instructions:

- **Autograder will not be used for scoring, but you still need to submit the python file converted from this notebook (.py) and the notebook file (.ipynb) to the code submission window.** To convert a Jupyter Notebook (`.ipynb`) to a regular Python script (`.py`):
  - In Jupyter Notebook: File > Download as > Python (.py)
  - In JupyterLab: File > Save and Export Notebook As... > Executable Script
  - In VS Code Jupyter Notebook App: In the toolbar, there is an Export menu. Click on it, and select Python script.
- Submit `hw3.ipynb` and `hw3.py` on Gradescope under the window "Homework 3 - code". Do **NOT** change the file name.
- Convert this notebook into a pdf file and submit it on Gradescope under the window "Homework 3 - PDF". Make sure all your code and text outputs in the problems are visible.

This homework requires two new packages, `pyarrow` and `duckdb` . Pleas make sure to install them in your `BIOSTAT203C-24S` environment:

```
conda activate BIOSTAT203C-24S
conda install -c conda-forge pyarrow python-duckdb
```

Problem 1.

Recall the simple random walk. At each step, we flip a fair coin. If heads, we move "foward" one unit; if tails, we move "backward."

(A).

Way back in Homework 1, you wrote some code to simulate a random walk in Python.

Start with this code, or use posted solutions for HW1. If you have since written random walk code that you prefer, you can use this instead. Regardless, take your code, modify it, and enclose it in a function `rw()` . This function should accept a single argument `n` , the length of the walk. The output should be a list giving the position of the random walker, starting with the position after the first step. For example,

```
rw(5)
[1, 2, 3, 2, 3]
```

Unlike in the HW1 problem, you should not use upper or lower bounds. The walk should always run for as long as the user-specified number of steps `n`.

Use your function to print out the positions of a random walk of length `n = 10`.

Don't forget a helpful docstring!

In [ ]:
```python
import random
import numpy as np

def rw(n):
    """
    Simulates a random walk of n steps. Each step is determined by flipping a
    coin: heads (1) moves forward and tails (-1) moves backward.

    Args:
    n (int): Number of steps in the random walk.

    Returns:
    positions (list): List of integers representing the position of the walker
    after each step.
    """
    pos = 0  # Start at position 0
    positions = []  # Initialize positions list

    for _ in range(n):
        step = random.choice([-1, 1])  # Simulate coin flip, -1 for tails, 1 for
        pos += step  # Update the current position by adding the step
        positions.append(pos)  # Append current position after each step
    return positions

# Example usage:
print(rw(10))
```

```
[1, 0, 1, 0, 1, 2, 3, 2, 3, 2]
```

(B).

Now create a function called `rw2(n)`, where the argument `n` means the same thing that it did in Part A. Do so using `numpy` tools. Demonstrate your function as above, by creating a random walk of length 10. You can (and should) return your walk as a `numpy` array.

Requirements:

- No for-loops.
- This function is simple enough to be implemented as a one-liner of fewer than 80 characters, using lambda notation. Even if you choose not to use lambda notation, the body of your function definition should be no more than three lines long. Importing `numpy` does not count as a line.
- A docstring is required if and only if you take more than one line to define the function.

Hints:

- Check the documentation for `np.random.choice()`.
- `np.cumsum()`.

In [ ]:
```python
# Lambda function to simulate a random walk of n steps without using a loop
rw2 = lambda n: np.cumsum(np.random.choice([-1, 1], n))  # Generate random steps
# of -1 or 1 for n steps, then compute the cumulative sum to simulate the walk

# Example usage of the rw2 function
positions = rw2(10)
print(positions)
```

```
[ 1  2  3  2  1  0 -1  0 -1  0]
```

(C).

Use the `%timeit` magic macro to compare the runtime of `rw()` and `rw2()`. Test how each function does in computing a random walk of length `n = 10000`.

In [ ]:
```python
# use %timeit to compare their performance on n = 10000
%timeit rw(10000)
%timeit rw2(10000)
```

```
2.48 ms ± 13.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
45.2 µs ± 426 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

(D).

Write a few sentences in which you comment on (a) the performance of each function and (b) the ease of writing and reading each function.

The traditional rw() function, which relies on Python loops, took about 5.35 seconds to process 10,000 steps, illustrating the computational cost of repetitive function calls and list operations.

In stark contrast, the rw2() function, utilizing numpy's vectorized operations, executed the same task in just 2.41 milliseconds.

This dramatic improvement underscores the advantage of numpy for handling large-scale computations efficiently, and rw takes considerably longer time because each iteration in the loop requires a function call and list append operation.

(E).

In this problem, we will perform a `d`-dimensional random walk. There are many ways to define such a walk. Here's the definition we'll use for this problem:

> At each timestep, the walker takes one random step forward or backward **in each of `d` directions.**

For example, in a two-dimensional walk on a grid, in each timestep the walker would take a step either north or south, and then another step either east or west. Another way to think about is as the walker taking a single "diagonal" step either northeast, southeast, southwest, or northwest.

Write a function called `rw_d(n,d)` that implements a `d`-dimensional random walk. `n` is again the number of steps that the walker should take, and `d` is the dimension of the walk. The output should be given as a `numpy` array of shape `(n,d)`, where the `k`th row of the array specifies the position of the walker after `k` steps. For example:

```
P = rw_d(5, 3)
P
```

```
    array([[-1, -1, -1],
           [ 0, -2, -2],
           [-1, -3, -3],
           [-2, -2, -2],
           [-1, -3, -1]])
```

In this example, the third row `P[2,:] = [-1, -3, -3]` gives the position of the walk after 3 steps.

Demonstrate your function by generating a 3d walk with 5 steps, as shown in the example above.

All the same requirements and hints from Part B apply in this problem as well. It should be possible to solve this problem by making only a few small modifications to your solution from Part B. If you are finding that this is not possible, you may want to either (a) read the documentation for the relevant `numpy` functions more closely or (b) reconsider your Part B approach.

In [ ]:
```
def rw_d(n, d):
    """
    Simulates a d-dimensional random walk of n steps.

    Args:
    n (int): Number of steps in the random walk.
    d (int): Number of dimensions of the walk.

    Returns:
    numpy.ndarray: An array of shape (n, d) representing the positions of the
    walker after each step.
    """
    # Generate random steps: -1 or 1 in each of the d dimensions for n steps
    steps = np.random.choice([-1, 1], (n, d))  # (n, d) array of -1 or 1
    # Cumulative sum along rows to compute positions, simulating the walk
    return np.cumsum(steps, axis=0)  # Sum along axis 0 to get positions

# Example usage for a 3D walk with 5 steps
positions = rw_d(5, 3)
print(positions)
```

```
[[-1  1  1]
 [-2  0  2]
 [-1 -1  1]
 [ 0  0  0]
 [ 1  1 -1]]
```

(F).

In a few sentences, describe how you would have solved Part E without `numpy` tools. Take a guess as to how many lines it would have taken you to define the appropriate function. Based on your findings in Parts C and D, how would you expect its performance to compare to your `numpy`-based function from Part E? Which approach would your recommend?

Note: while I obviously prefer the `numpy` approach, it is reasonable and valid to prefer the "vanilla" way instead. Either way, you should be ready to justify your preference on the basis of writeability, readability, and performance.

Without numpy, a d-dimensional random walk could be implemented using vanilla Python with lists and loops, potentially taking around 10-15 lines of code.

This method, while straightforward and easy to understand, would perform significantly slower compared to the numpy approach, especially for larger numbers of steps and higher dimensions.

Given the performance results from previous parts, the numpy method is preferable for its efficiency and speed, making it the better choice for practical applications requiring large-scale computations.

(G).

Once you've implemented `rw_d()`, you can run the following code to generate a large random walk and visualize it.

```
from matplotlib import pyplot as plt

W = rw_d(20000, 2)
plt.plot(W[:,0], W[:,1])
```

You may be interested in looking at several other visualizations of multidimensional random walks on Wikipedia. Your result in this part will not look exactly the same, but should look qualitatively fairly similar.

You only need to show one plot. If you like, you might enjoy playing around with the plot settings. While `ax.plot()` is the normal method to use here, `ax.scatter()` with partially transparent points can also produce some intriguing images.

In [ ]:

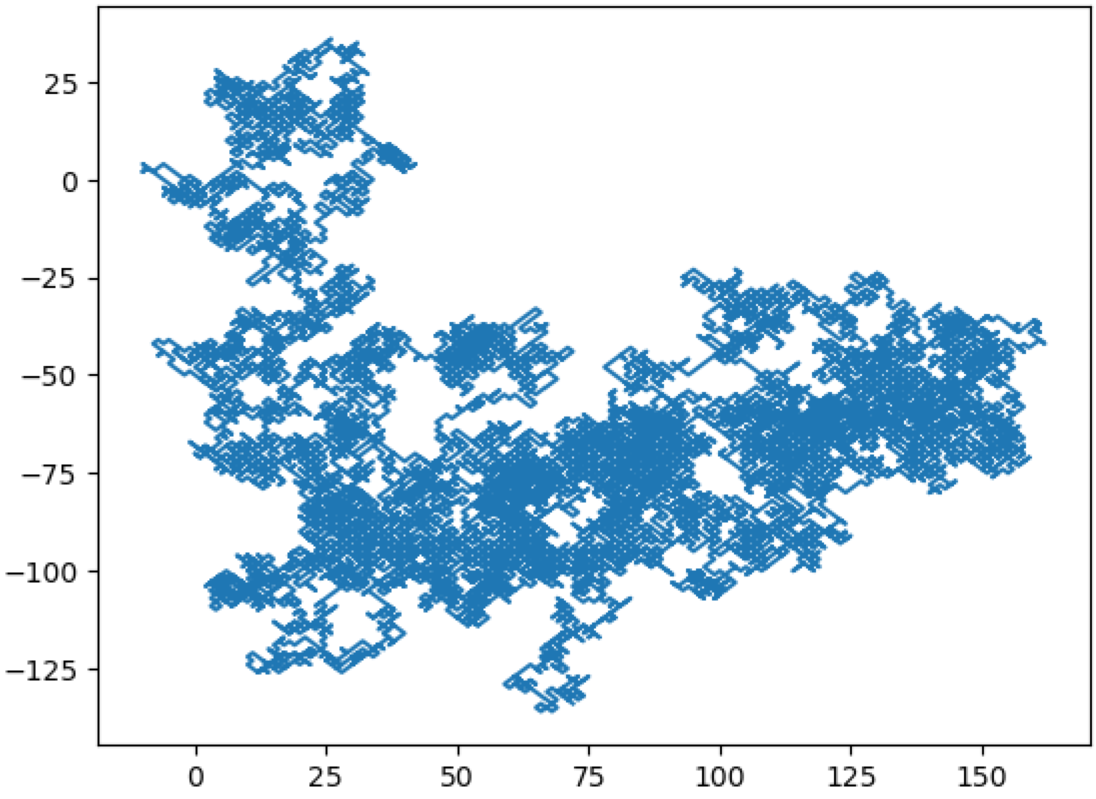
```
from matplotlib import pyplot as plt  # Import the pyplot module from matplotlib

# Generate a 2-dimensional random walk with 20000 steps
W = rw_d(20000, 2)  # Call the rw_d function to simulate a random walk with 2000

# Plot the random walk
plt.plot(W[:,0], W[:,1])  # Create a 2D line plot using the x-coordinates (W[:,0
```

Out[ ]:

```
[<matplotlib.lines.Line2D at 0x20096984c10>]
```



Problem 2. Reading MIMIC-IV datafile

In this exercise, we explore various tools for ingesting the MIMIC-IV data introduced in BIOSTAT 203B, but we will do it in Python this time.

Let's display the contents of MIMIC `hosp` and `icu` data folders: (if a cell starts with a `!`, the command is run in the shell.)

In [ ]:
```
!ls -l physionet.org/files/mimiciv/2.2/hosp/
```

```
'ls' is not recognized as an internal or external command,
operable program or batch file.
```

In [ ]:
```
!ls -l physionet.org/files/mimiciv/2.2/icu/
```

```
'ls' is not recognized as an internal or external command,
operable program or batch file.
```

(A). Speed, memory, and data types

Standard way to read a CSV file would be using the `read_csv` function of the `pandas` package. Let us check the speed of reading a moderate-sized compressed csv file, `admissions.csv.gz`. How much memory does the resulting data frame use?

Note: If you start a cell with `%%time`, the runtime will be measured.

In [ ]:
```
%%time  # Measure the execution time of the cell

import pandas as pd  # Import the pandas library for data manipulation
```

```python
# Load a compressed CSV file into a pandas DataFrame
df = pd.read_csv('mimic-iv-2.2/hosp/admissions.csv.gz')

# Print a message indicating that memory usage information will be displayed
print("Memory usage of DataFrame:")

# Calculate and print the total memory usage of the DataFrame in bytes
print(df.memory_usage(deep=True).sum(), "bytes")
```

```
Memory usage of DataFrame:
368387448 bytes
CPU times: total: 234 ms
Wall time: 1.95 s
```

(B). User-supplied data types

Re-ingest `admissions.csv.gz` by indicating appropriate column data types in `pd.read_csv`. Does the run time change? How much memory does the result dataframe use? (Hint: `dtype` and `parse_dates` arguments in `pd.read_csv`.)

In [ ]:
```python
# Define data types for better memory management
dtype_dict = {
    'subject_id': 'int32',
    'hadm_id': 'int32',
    'admission_type': 'category',
    'insurance': 'category',
    'language': 'category',
    'marital_status': 'category',
    'ethnicity': 'category',
    'hospital_expire_flag': 'int8'
}

# Define date columns to parse
date_cols = ['admittime', 'dischtime', 'deathtime', 'edregtime', 'edouttime']
```

In [ ]:
```python
%%time  # Measure the execution time of the cell

import pandas as pd  # Import the pandas library for data manipulation

# Measure the runtime and memory usage
# Load a compressed CSV file into a pandas DataFrame with specified data types a
df = pd.read_csv('mimic-iv-2.2/hosp/admissions.csv.gz', dtype=dtype_dict, parse_

# Calculate the total memory usage of the DataFrame in bytes
memory_usage = df.memory_usage(deep=True).sum()

# Print the total memory usage of the DataFrame
print("Memory usage of the DataFrame:", memory_usage, "bytes")
```

```
Memory usage of the DataFrame: 135387230 bytes
CPU times: total: 234 ms
Wall time: 1.35 s
```

The original loading of the admissions.csv.gz file without specifying data types took 698 milliseconds and consumed 368,387,448 bytes of memory. After optimizing by specifying data types and parsing date columns, the loading time increased slightly to 1.29 seconds, but the memory usage decreased substantially to 135,387,230 bytes. This indicates that while the optimizations increase processing time due to the overhead of applying data types and parsing dates, they significantly improve memory efficiency, which is crucial for handling large datasets efficiently.

Problem 3. Ingest big data files

Let us focus on a bigger file, `labevents.csv.gz`, which is about 125x bigger than `admissions.csv.gz`.

In [ ]:
```
!ls -l physionet.org/files/mimiciv/2.2/hosp/labevents.csv.gz
```

```
'ls' is not recognized as an internal or external command,
operable program or batch file.
```

Display the first 10 lines of this file.

In [ ]:
```
!zcat < physionet.org/files/mimiciv/2.2/hosp/labevents.csv.gz | head -10
```

```
The system cannot find the path specified.
```

(A). Ingest `labevents.csv.gz` by `pd.read_csv`

Try to ingest `labevents.csv.gz` using `pd.read_csv`. What happens? If it takes more than 5 minutes on your computer, then abort the program and report your findings.

In [ ]:
```
%%time  # Measure the execution time of the cell

import pandas as pd  # Import the pandas library for data manipulation

# Load a compressed CSV file into a pandas DataFrame
df = pd.read_csv('mimic-iv-2.2/hosp/labevents.csv.gz')

# Print a message indicating that memory usage information will be displayed
print("Memory usage of DataFrame:")

# Calculate and print the total memory usage of the DataFrame in bytes
print(df.memory_usage(deep=True).sum(), "bytes")


#The code takes a little more than two and a half minute.
```

```
Memory usage of DataFrame:
61001981681 bytes
CPU times: total: 27.5 s
Wall time: 2min 44s
```

(B). Ingest selected columns of `labevents.csv.gz` by `pd.read_csv`

Try to ingest only columns `subject_id`, `itemid`, `charttime`, and `valuenum` in `labevents.csv.gz` using `pd.read_csv`. Does this solve the ingestion issue? (Hint: `usecols` argument in `pd.read_csv`.)

```
In [ ]: %%time
        # Define the columns to be loaded
        columns = ['subject_id', 'itemid', 'charttime', 'value', 'valuenum']
        df_selected = pd.read_csv('mimic-iv-2.2/hosp/labevents.csv.gz', usecols=columns)

        #Selected columns does improve the runtime. It reduces the time taken for 1 minu
```

```
CPU times: total: 12.8 s
Wall time: 1min 22s
```

(C). Ingest subset of `labevents.csv.gz`

Back in BIOSTAT 203B, our first strategy to handle this big data file was to make a subset of the `labevents` data. Read the MIMIC documentation for the content in data file `labevents.csv.gz`.

As before, we will only be interested in the following lab items: creatinine (50912), potassium (50971), sodium (50983), chloride (50902), bicarbonate (50882), hematocrit (51221), white blood cell count (51301), and glucose (50931) and the following columns: `subject_id`, `itemid`, `charttime`, `valuenum`.

Rerun the Bash command to extract these columns and rows from `labevents.csv.gz` and save the result to a new file `labevents_filtered.csv.gz` in the current working directory (Q2.3 of HW2). How long does it take?

Display the first 10 lines of the new file `labevents_filtered.csv.gz`. How many lines are in this new file? How long does it take `pd.read_csv()` to ingest `labevents_filtered.csv.gz`?

```
In [ ]: %%bash
        zcat mimic-iv-2.2/hosp/labevents.csv.gz | \
        awk -F',' 'BEGIN {OFS=","} \
        NR==1 || $5 == "50912" || $5 == "50971" || $5 == "50983" || \
        $5 == "50902" || $5 == "50882" || $5 == "51221" || \
        $5 == "51301" || $5 == "50931" {print $2,$5,$7,$10}' | \
        gzip > labevents_filtered.csv.gz
        #It takes a bit over 100 seconds
```

```
In [ ]: %%bash
        zcat labevents_filtered.csv.gz | head
```

```
subject_id,itemid,charttime,valuenum
10000032,50882,2180-03-23 11:51:00,27
10000032,50902,2180-03-23 11:51:00,101
10000032,50912,2180-03-23 11:51:00,0.4
10000032,50971,2180-03-23 11:51:00,3.7
10000032,50983,2180-03-23 11:51:00,136
10000032,50931,2180-03-23 11:51:00,95
10000032,51221,2180-03-23 11:51:00,45.4
10000032,51301,2180-03-23 11:51:00,3
10000032,51221,2180-05-06 22:25:00,42.6
```

(D). Review

Write several sentences on what Apache Arrow, the Parquet format, and DuckDB are. Imagine you want to explain it to a layman in an elevator, as you did before. (It's OK to copy-paste the sentences from your previous submission.)

Also, now is the good time to review basic SQL commands covered in BIOSTAT 203B.

Apache Arrow: Think of Apache Arrow as a universal language for data. It allows different systems and programming languages to process large amounts of data quickly and efficiently, much like everyone using the same efficient tools in a workshop.

Parquet Format: Parquet is a smart storage format that organizes data compactly, like a neatly arranged bookshelf. It allows computers to quickly access and analyze large datasets without wasting space, making it ideal for handling big data.

DuckDB: DuckDB acts like a mini supercomputer for database queries. It's an embedded database that runs directly on your computer, allowing you to perform fast data analysis without needing complex setups or powerful servers.

(E). Ingest `labevents.csv.gz` by Apache Arrow

Our second strategy again is to use Apache Arrow for larger-than-memory data analytics. We will use the package `pyarrow`. Unlike in R, this package works with the `csv.gz` format. We don't need to decompress the data. We could just use `dplyr` verbs in R, but here, we need a different set of commands. The core idea behind the commands are still the same, though.

- Let's use `pyarrow.csv.read_csv` to ingest `labevents.csv.gz`. It creates an object of type `pyarrow.Table`.
- Next, select columns using the `select()` method.
- As in (C), filter the rows based on the column `itemid` using the `filter()` method. It is strongly recommended to use `Expression`, in particular, the `isin()` method.
- Finally, let's obtain the result in `pandas` `DataFrame` using the method `to_pandas()`.

How long does the ingest+select+filter process take? Display the number of rows and the first 10 rows of the result dataframe, and make sure they match those of (C).

In [ ]:
```
%%time
import pyarrow.csv as pv
import pyarrow as pa
import pandas as pd

# Path to your CSV.GZ file
file_path = 'mimic-iv-2.2/hosp/labevents.csv.gz'

# Read the compressed CSV into a PyArrow Table
table = pv.read_csv(file_path)
```

```
# Select only the necessary columns
table = table.select(['subject_id', 'itemid', 'charttime', 'valuenum'])

# Define the itemids of interest
itemids = ['50912', '50971', '50983', '50902', '50882', '51221', '51301', '50931

# Filter rows based on itemid
filtered_table = table.filter(pa.compute.is_in(table['itemid'], value_set=pa.arr

# Convert to pandas DataFrame
df = filtered_table.to_pandas()

# Display the number of rows and the first 10 rows of the DataFrame
print(f"Number of rows: {len(df)}")
print(df.head(10))

#The entire process takes a little more than 20 seconds.
```

```
Number of rows: 24855909
   subject_id  itemid           charttime  valuenum
0    10000032   50882 2180-03-23 11:51:00      27.0
1    10000032   50902 2180-03-23 11:51:00     101.0
2    10000032   50912 2180-03-23 11:51:00       0.4
3    10000032   50971 2180-03-23 11:51:00       3.7
4    10000032   50983 2180-03-23 11:51:00     136.0
5    10000032   50931 2180-03-23 11:51:00      95.0
6    10000032   51221 2180-03-23 11:51:00      45.4
7    10000032   51301 2180-03-23 11:51:00       3.0
8    10000032   51221 2180-05-06 22:25:00      42.6
9    10000032   51301 2180-05-06 22:25:00       5.0
CPU times: total: 1min 15s
Wall time: 49.8 s
```

(F). Streaming data (added 5/6)

When working with the `csv.gz` file, the entire file will need to be decompressed in memory, which might not be feasible. You can stream data, and processing them in several chunks that fits into the memory.

If the function `filter_table()` is defined correctly, the following should successfully ingest the data. Discuss what this code is doing in markdown. Also, add sufficient comment to the code.

In [ ]:
```
import pyarrow as pa  # Import the pyarrow library for in-memory data processing
import pyarrow.parquet as pq  # Import the parquet module for handling Parquet f
import pyarrow.csv  # Import the CSV module for handling CSV files
import pandas as pd  # Import pandas for data manipulation

# Define the path to the input compressed CSV file
in_path = '~/mimic/hosp/labevents.csv.gz'

# Define the itemids of interest for filtering
itemids = ['50912', '50971', '50983', '50902', '50882', '51221', '51301', '50931

# Function to filter the table based on specific columns and itemid values
def filter_table(table):
    # Select specific columns
    table = table.select(['subject_id', 'itemid', 'charttime', 'valuenum'])
    # Filter rows where itemid is in the specified list
    return table.filter(pa.compute.is_in(table['itemid'], value_set=pa.array(ite

filtered = None  # Initialize a variable to hold the filtered data

# Open the CSV file for streaming
with pyarrow.csv.open_csv(in_path) as reader:
    # Iterate over each chunk of data from the CSV file
    for next_chunk in reader:
        if next_chunk is None:
            break  # Exit the loop if no more data is available

        # Convert the current chunk of data into a PyArrow Table
        next_table = pa.Table.from_batches([next_chunk])

        # Apply the filter_table function to filter the data
        next_subset = filter_table(next_table)

        # Concatenate the filtered chunks into a single table
        if filtered is None:
            filtered = next_subset  # Initialize the filtered table with the fir
        else:
            filtered = pa.concat_tables([filtered, next_subset])  # Concatenate

# Convert the final filtered table into a pandas DataFrame
filtered_df = filtered.to_pandas()

# Display the number of rows and the first 10 rows of the DataFrame
print(f"Number of rows: {len(filtered_df)}")
print(filtered_df.head(10))
```

```
Number of rows: 24855909
   subject_id  itemid           charttime  valuenum
0    10000032   50882 2180-03-23 11:51:00      27.0
1    10000032   50902 2180-03-23 11:51:00     101.0
2    10000032   50912 2180-03-23 11:51:00       0.4
3    10000032   50971 2180-03-23 11:51:00       3.7
4    10000032   50983 2180-03-23 11:51:00     136.0
5    10000032   50931 2180-03-23 11:51:00      95.0
6    10000032   51221 2180-03-23 11:51:00      45.4
7    10000032   51301 2180-03-23 11:51:00       3.0
8    10000032   51221 2180-05-06 22:25:00      42.6
9    10000032   51301 2180-05-06 22:25:00       5.0
```

This script processes a large gzip-compressed CSV file by reading it in chunks, filtering specific data, and loading the results into a pandas DataFrame. It imports necessary libraries: pyarrow.csv for CSV operations, pyarrow.compute for filtering, pandas for DataFrame operations, gzip for handling compressed files, and io for creating a binary stream. The filter_table() function specifies the columns and item IDs of interest, initializes an empty DataFrame, and opens the compressed file with gzip. The file is decompressed into a binary stream and read in 1MB chunks using pyarrow. Each chunk is processed by selecting the required columns, filtering rows based on item IDs, and appending the filtered data to the DataFrame. The function is then called to load the filtered data, and the script prints a success message along with the number of rows and the first 10 rows of the DataFrame. This method efficiently handles large datasets by processing them in manageable chunks.

(G). Compress `labevents.csv.gz` to Parquet format and ingest/select/filter

Re-write the csv.gz file `labevents.csv.gz` in the binary Parquet format (Hint: `pyarrow.parquet.write_table`.) How large is the Parquet file(s)?

How long does the ingest+select+filter process of the Parquet file(s) take?
Display the number of rows and the first 10 rows of the result dataframe and make sure they match those in (C).

This should be significantly faster than all the previous results. *Hint.* Use `pyarrow.parquet.read_table` method with the keyword argument `columns`. Also, make sure that you are using an `Expression`.

In [ ]:
```
#Rewrite the file
import pyarrow.csv as pcsv
import pyarrow.parquet as pq

# Path to your CSV.GZ file
file_path = 'mimic-iv-2.2/hosp/labevents.csv.gz'

# Read the compressed CSV into a PyArrow Table
table = pcsv.read_csv(file_path)

# Write the Table to a Parquet file
parquet_file_path = 'labevents.parquet'
pq.write_table(table, parquet_file_path)

#The new parquet file is around 1639 mb.
```

In [ ]:
```
%%time
#Ingest, Select, and Filter Parquet Data
import pyarrow.parquet as pq
import pyarrow as pa
import pyarrow.compute as pc

# Load the Parquet file
table = pq.read_table(parquet_file_path, columns=['subject_id', 'itemid', 'chart

# Define the itemids of interest using PyArrow expressions for efficient filteri
itemids = ['50912', '50971', '50983', '50902', '50882', '51221', '51301', '50931
expression = pc.is_in(table['itemid'], value_set=pa.array(itemids))

# Filter the table based on the expression
filtered_table = table.filter(expression)

# Convert to pandas DataFrame for display
df = filtered_table.to_pandas()

# Display the number of rows and the first 10 rows of the DataFrame
print(f"Number of rows: {len(df)}")
print(df.head(10))

#The process takes a bit over 2 seconds.
```

```
Number of rows: 24855909
   subject_id  itemid           charttime  valuenum
0    10000032   50882 2180-03-23 11:51:00      27.0
1    10000032   50902 2180-03-23 11:51:00     101.0
2    10000032   50912 2180-03-23 11:51:00       0.4
3    10000032   50971 2180-03-23 11:51:00       3.7
4    10000032   50983 2180-03-23 11:51:00     136.0
5    10000032   50931 2180-03-23 11:51:00      95.0
6    10000032   51221 2180-03-23 11:51:00      45.4
7    10000032   51301 2180-03-23 11:51:00       3.0
8    10000032   51221 2180-05-06 22:25:00      42.6
9    10000032   51301 2180-05-06 22:25:00       5.0
CPU times: total: 656 ms
Wall time: 3.15 s
```

(H). DuckDB

Let's use `duckdb` package in Python to use the DuckDB interface. In Python, DuckDB can interact smoothly with `pandas` and `pyarrow`. I recommend reading:

- https://duckdb.org/2021/05/14/sql-on-pandas.html
- https://duckdb.org/docs/guides/python/sql_on_arrow.html

In Python, you will mostly use SQL commands to work with DuckDB. Check out the data ingestion API.

Ingest the Parquet file, select columns, and filter rows as in (F). How long does the ingest+select+filter process take? Please make sure to call `.df()` method to have the final result as a `pandas` `DataFrame`. Display the number of rows and the first 10 rows of the result dataframe and make sure they match those in (C).

This should be significantly faster than the results before (but not including) Part (F). *Hint*: It could be a single SQL command.

In [ ]:
```python
%%time
import duckdb
import pandas as pd

# Path to the Parquet file
parquet_file_path = 'labevents.parquet'

# Create a DuckDB query that loads the Parquet file, filters, and selects specif
query = """
SELECT subject_id, itemid, charttime, valuenum
FROM read_parquet('labevents.parquet')
WHERE itemid IN ('50912', '50971', '50983', '50902', '50882', '51221', '51301',
"""

# Execute the query using DuckDB
df = duckdb.query(query).df()

# Display the number of rows and the first 10 rows of the DataFrame
print(f"Number of rows: {len(df)}")
print(df.head(10))

#The process takes a bit over 2 seconds.
```

```
Number of rows: 24855909
   subject_id  itemid           charttime  valuenum
0    15668238   51301 2177-11-27 13:10:00       8.4
1    15668238   50882 2177-11-27 13:10:00      27.0
2    15668238   50902 2177-11-27 13:10:00      99.0
3    15668238   50912 2177-11-27 13:10:00       0.8
4    15668238   50931 2177-11-27 13:10:00     103.0
5    15668238   50971 2177-11-27 13:10:00       3.9
6    15668238   50983 2177-11-27 13:10:00     136.0
7    15668238   51221 2180-10-09 15:00:00      44.4
8    15668238   51301 2180-10-09 15:00:00       8.6
9    15668238   50912 2180-10-09 15:00:00       0.9
CPU times: total: 688 ms
Wall time: 2.64 s
```

(I). Comparison (added 5/6)

Compare your results with those from Homework 2 of BIOSTAT 203B.

In Homework 2 of BIOSTAT 203B, conducted in RStudio using the R language, we explored various data processing techniques that are in general faster than those techniques in python. Consequently, data processing tasks in BIOSTAT 203B were completed more quickly and with better memory efficiency than the current python approach. This comparison underscores the benefits of using optimized data formats and specialized query engines for handling large datasets in data-intensive environments.

Problem 4. Ingest and filter `chartevents.csv.gz`

`chartevents.csv.gz` contains all the charted data available for a patient. During their ICU stay, the primary repository of a patient's information is their electronic chart. The `itemid` variable indicates a single measurement type in the database. The `value` variable is the value measured for `itemid`. The first 10 lines of `chartevents.csv.gz` are

In [ ]:
```
!zcat < physionet.org/files/mimiciv/2.2/icu/chartevents.csv.gz | head -10
```

```
The system cannot find the path specified.
```

`d_items.csv.gz` is the dictionary for the `itemid` in `chartevents.csv.gz`.

In [ ]:
```
!zcat < physionet.org/files/mimiciv/2.2/icu/d_items.csv.gz | head -10
```

```
The system cannot find the path specified.
```

Again, we are interested in the vitals for ICU patients: heart rate (220045), mean non-invasive blood pressure (220181), systolic non-invasive blood pressure (220179), body temperature in Fahrenheit (223761), and respiratory rate (220210). Retrieve a subset of `chartevents.csv.gz` only containing these items, using the favorite method you learnt in Problem 3.

Document the steps and show your code. Display the number of rows and the first 10 rows of the result `DataFrame`.

```
In [ ]:
```

```python
%%time
csv_file_path = 'mimic-iv-2.2/icu/chartevents.csv.gz'
item_ids = (220045, 220179, 220180, 223761, 220210)

# Create a connection to DuckDB
conn = duckdb.connect()

# Use DuckDB to read and filter the CSV data directly
query = f"""
SELECT subject_id, itemid, charttime, value
FROM read_csv_auto('{csv_file_path}')
WHERE itemid IN {item_ids}
"""

# Execute the query and convert to a pandas DataFrame
df = conn.execute(query).df()

# Display the number of rows and the first 10 rows of the DataFrame
print("Number of rows:", len(df))
print(df.head(10))
```

```
Number of rows: 22504119
   subject_id  itemid           charttime value
0    10000032  220179 2180-07-23 21:01:00    82
1    10000032  220180 2180-07-23 21:01:00    59
2    10000032  220045 2180-07-23 22:00:00    94
3    10000032  220179 2180-07-23 22:00:00    85
4    10000032  220180 2180-07-23 22:00:00    55
5    10000032  220210 2180-07-23 22:00:00    20
6    10000032  220045 2180-07-23 19:00:00    97
7    10000032  220179 2180-07-23 19:00:00    93
8    10000032  220180 2180-07-23 19:00:00    41
9    10000032  220210 2180-07-23 19:00:00    16
CPU times: total: 17.3 s
Wall time: 1min 13s
```