Medical Insurance Database System Simulation with Operating System Concepts

Daisy Aptovska

Mariami Shinjiashvili

CMPSC 472: Operating Systems Concepts

Dr. Janghoon Yang

December 10, 2025

Table of Contents

Description of the Project

In our project, we utilized a publicly available dataset of medical insurance customers, their demographics, and their medical insurance cost. We simulated the process of a healthcare database system by querying this dataset, sorting the medical insurance cost, and creating graphical displays of the medial insurance data by demographic. Through this system-based application, we utilized multithreading to create time comparisons on different number of threads on a certain process (querying, sorting, and visualizing the medical insurance data). The aim of this project is to demonstrate the importance of the threads on a system-level application and how they can speed up the operations within a large data process.

To utilize our project, follow these steps:

1. Download all 4 Python files from our GitHub repository at the following URL: https://github.com/daisyapto/CMPSC472_Project2
2. Download the Train_Data.csv file from the Kaggle dataset we used at the following URL: https://www.kaggle.com/datasets/gauravduttakiit/medical-insurance-cost/data?select=Train_Data.csv
3. Run the main.py file to run the program and interact with the user interface

Significance of the Project

Our project's significance is it aims to provide efficiency within medical insurance companies' data processing. Many insurance companies are quite large, consisting of hundreds or thousands of clients. It is important that when querying, sorting, and visualizing their data that they have an efficient system-level application to assist with this process. While many database systems use SQL and a simple querying method, it is important to emphasize the use of multithreading to create efficiency in access to large dataset. Rather than using a SQL approach, this medical insurance database simulation system uses Python multi-threading with the Pandas data library for reading and display manipulation.

Code Structure

In our code structure, we modularized the code into 3 core class. We created a sorting class to execute selection sort and insertion sort on a given data segment of the medical insurance charges column of our data. We created a visualization class which contains 4 types of functions: editing bar, pie, or line chart objects and helper functions that call the editing functions for the type of visualization chosen. Lastly, we made a class for the threading functions. In this class, the backend functionality of thread creation is made based on the functionality being used. Within our main function, we created the framing of the GUI, and in the main file, we created the functionality for searching and threading. Figure 1 shows the UML diagram of our program, and Figure 2 shows the file structure of our project.

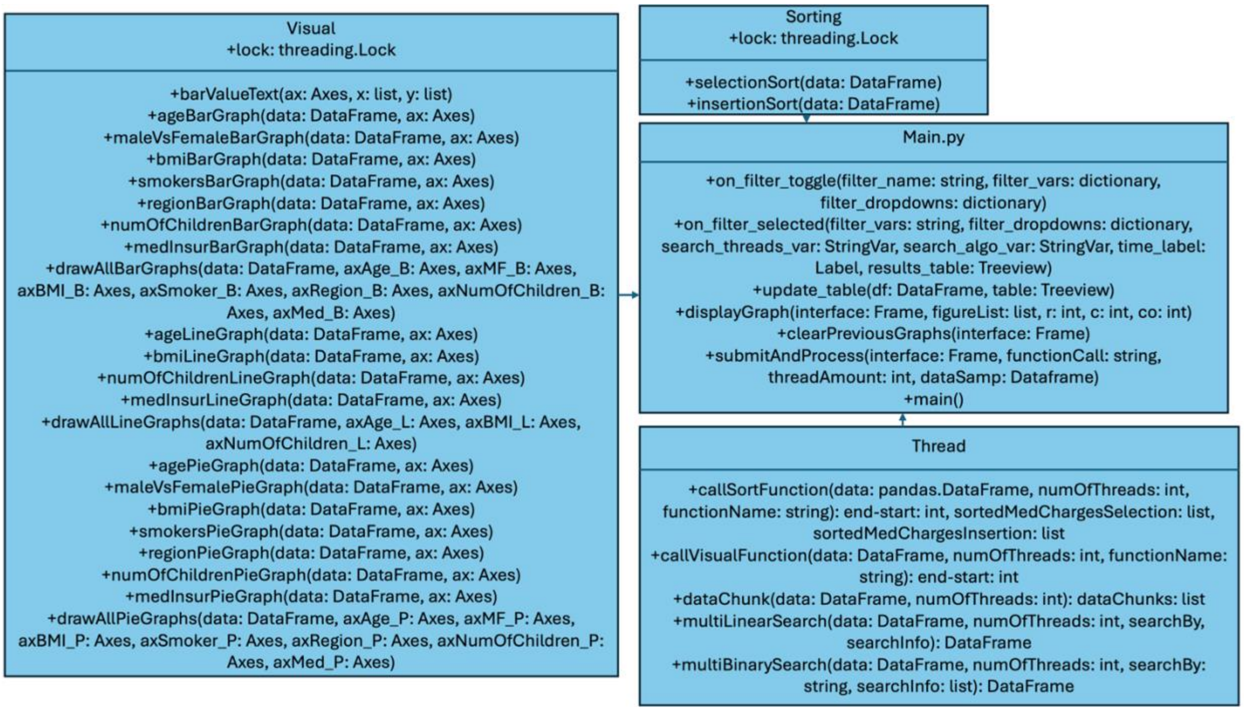

Visual
+lock: threading.Lock
+barValueText(ax: Axes, x: list, y: list)
+ageBarGraph(data: DataFrame, ax: Axes)
+maleVsFemaleBarGraph(data: DataFrame, ax: Axes)
+bmiBarGraph(data: DataFrame, ax: Axes)
+smokersBarGraph(data: DataFrame, ax: Axes)
+regionBarGraph(data: DataFrame, ax: Axes)
+numOfChildrenBarGraph(data: DataFrame, ax: Axes)
+medInsurBarGraph(data: DataFrame, ax: Axes)
+drawAllBarGraphs(data: DataFrame, axAge_B: Axes, axMF_B: Axes, axBMI_B: Axes, axSmoker_B: Axes, axRegion_B: Axes, axNumOfChildren_B: Axes, axMed_B: Axes)
+ageLineGraph(data: DataFrame, ax: Axes)
+bmiLineGraph(data: DataFrame, ax: Axes)
+numOfChildrenLineGraph(data: DataFrame, ax: Axes)
+medInsurLineGraph(data: DataFrame, ax: Axes)
+drawAllLineGraphs(data: DataFrame, axAge_L: Axes, axBMI_L: Axes, axNumOfChildren_L: Axes)
+agePieGraph(data: DataFrame, ax: Axes)
+maleVsFemalePieGraph(data: DataFrame, ax: Axes)
+bmiPieGraph(data: DataFrame, ax: Axes)
+smokersPieGraph(data: DataFrame, ax: Axes)
+regionPieGraph(data: DataFrame, ax: Axes)
+numOfChildrenPieGraph(data: DataFrame, ax: Axes)
+medInsurPieGraph(data: DataFrame, ax: Axes)
+drawAllPieGraphs(data: DataFrame, axAge_P: Axes, axMF_P: Axes, axBMI_P: Axes, axSmoker_P: Axes, axRegion_P: Axes, axNumOfChildren_P: Axes, axMed_P: Axes)
Sorting
+lock: threading.Lock
+selectionSort(data: DataFrame)
+insertionSort(data: DataFrame)
Main.py
+on_filter_toggle(filter_name: string, filter_vars: dictionary, filter_dropdowns: dictionary)
+on_filter_selected(filter_vars: string, filter_dropdowns: dictionary, search_threads_var: StringVar, search_algo_var: StringVar, time_label: Label, results_table: Treeview)
+update_table(df: DataFrame, table: Treeview)
+displayGraph(interface: Frame, figureList: list, r: int, c: int, co: int)
+clearPreviousGraphs(interface: Frame)
+submitAndProcess(interface: Frame, functionCall: string, threadAmount: int, dataSamp: Dataframe)
+main()
Thread
+callSortFunction(data: pandas.DataFrame, numOfThreads: int, functionName: string): end-start: int, sortedMedChargesSelection: list, sortedMedChargesInsertion: list
+callVisualFunction(data: DataFrame, numOfThreads: int, functionName: string): end-start: int
+dataChunk(data: DataFrame, numOfThreads: int): dataChunks: list
+multiLinearSearch(data: DataFrame, numOfThreads: int, searchBy, searchInfo): DataFrame
+multiBinarySearch(data: DataFrame, numOfThreads: int, searchBy: string, searchInfo: list): DataFrame

Figure 1: UML Diagram

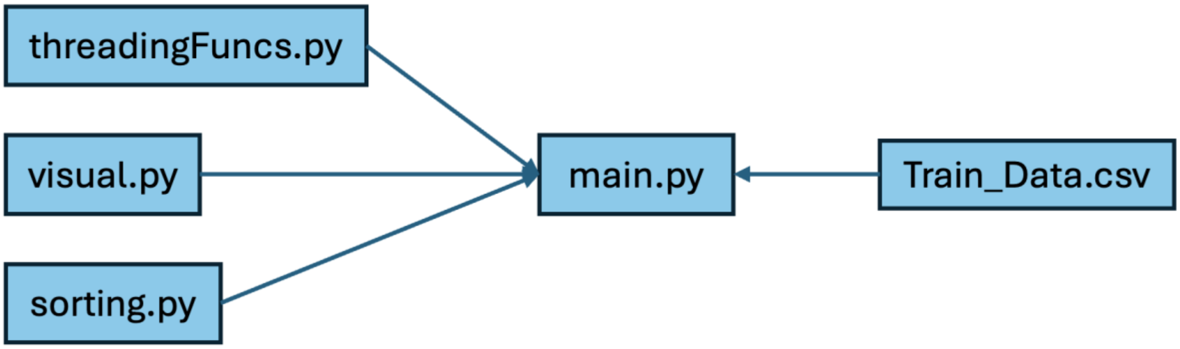


Figure 2: File Structure

Description of Algorithms

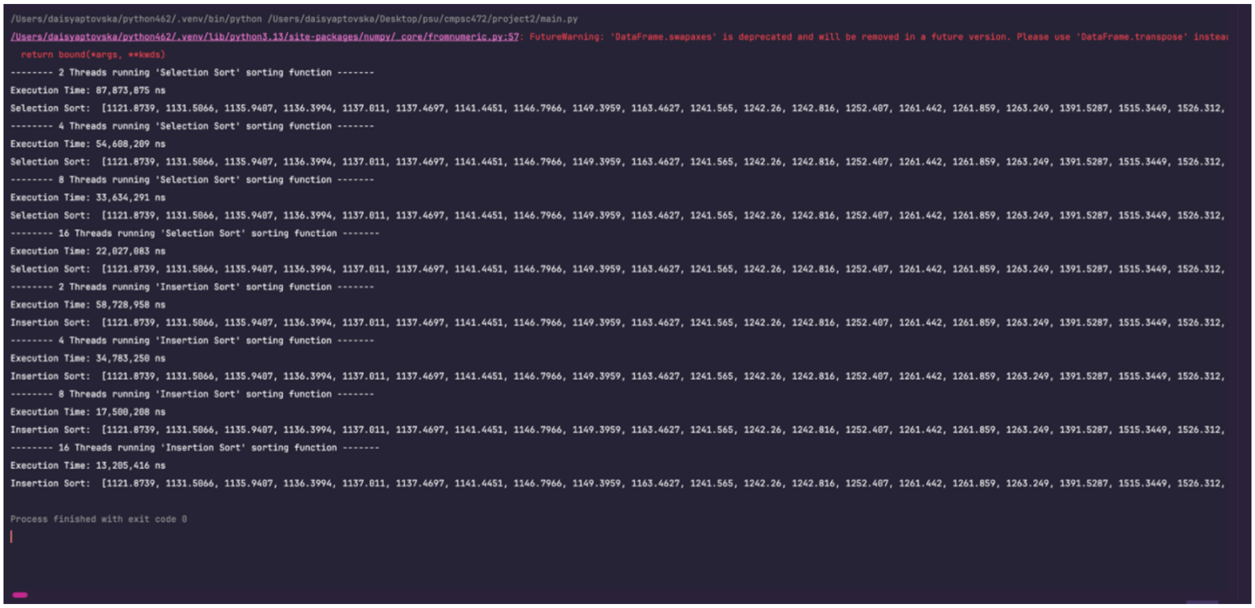
For the querying of the data, we implemented a linear search algorithm and a binary search algorithm to test how more threads will execute this process faster. We created a filter menu to allow users to choose which column category they were querying in the dataset. For example, users can choose the find all patients in the database who are of age 45, or they can find all patients with a medical insurance charge of $5,000. This filter menu adds much versatility to the querying process. Users choose the number of threads they are executing their linear or binary search of the data with. Under the display of the queried data, an execution time is shown to exemplify the importance of threads and which algorithm is used to optimize the execution time.

For the sorting of the data, we tested two different sorting methods on the medical insurance charges column of the dataset: selection sort and insertion sort. After isolating the selection sort and insertion sort process with the number of threads chosen by the user, the sorted data table is displayed. We verify the sorted output of selection sort and insertion sort as a list in the terminal. The execution time for sorting only includes the backend sorting process that is shown in the terminal.

For our visualization of the data, we used Matplotlib to visualize all the data in the dataset using different thread counts. One of our limitations (discussed further in Functionalities section) originally was that Tkinter and Matplotlib are not thread safe by default. We used locks in the visualization functions to update the plot objects. Then, once we ensured the object was safely updated, we could display the plot. We found the most difficulty with the line chart display since they required a sample of the dataset and sorted parallel lists for display. We sampled the dataset outside of the main routine to ensure that the dataset is only sampled once, and the same sample is used for all line charts. An issue we ran into was that instead of using the same sample, the program would use a new sample every time, plotting multiple lines onto the same plot. Now, since the same sample is used from outside the main routine, the same graph is overwrote onto the same plot, and the time can be calculated this way every time line chart is called.

Verification of Algorithms

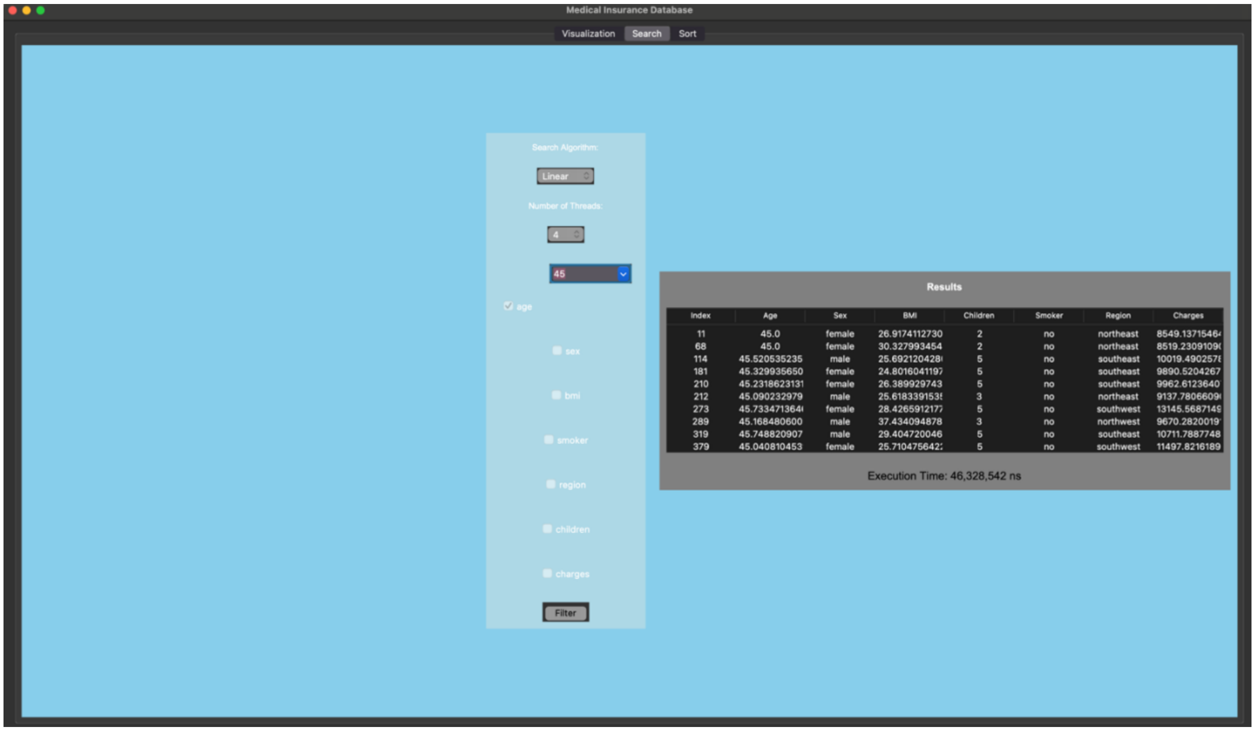
In the verification process of our algorithms, we tested if the output was sorted through debugging and printing our output. Once we ensured that the output was sorted, we used a heap merge function to merge the number of data chunks used into merged sorted lists. The number of data chunks was dependent on the number of threads used.



```
/Users/daisyaptovska/python462/.venv/bin/python /Users/daisyaptovska/Desktop/psu/cmpsc472/project2/main.py
/Users/daisyaptovska/python462/.venv/lib/python3.13/site-packages/numpy/_core/fromnumeric.py:57: FutureWarning: 'DataFrame.swapaxes' is deprecated and will be removed in a future version. Please use 'DataFrame.transpose' instea
  return bound(*args, **kwds)
-------- 2 Threads running 'Selection Sort' sorting function -------
Execution Time: 87,873,875 ns
Selection Sort:  [1121.8739, 1131.5066, 1135.9407, 1136.3994, 1137.011, 1137.4697, 1141.4451, 1146.7966, 1149.3959, 1163.4627, 1241.565, 1242.26, 1242.816, 1252.407, 1261.442, 1261.859, 1263.249, 1391.5287, 1515.3449, 1526.312,
-------- 4 Threads running 'Selection Sort' sorting function -------
Execution Time: 54,608,209 ns
Selection Sort:  [1121.8739, 1131.5066, 1135.9407, 1136.3994, 1137.011, 1137.4697, 1141.4451, 1146.7966, 1149.3959, 1163.4627, 1241.565, 1242.26, 1242.816, 1252.407, 1261.442, 1261.859, 1263.249, 1391.5287, 1515.3449, 1526.312,
-------- 8 Threads running 'Selection Sort' sorting function -------
Execution Time: 33,634,291 ns
Selection Sort:  [1121.8739, 1131.5066, 1135.9407, 1136.3994, 1137.011, 1137.4697, 1141.4451, 1146.7966, 1149.3959, 1163.4627, 1241.565, 1242.26, 1242.816, 1252.407, 1261.442, 1261.859, 1263.249, 1391.5287, 1515.3449, 1526.312,
-------- 16 Threads running 'Selection Sort' sorting function -------
Execution Time: 22,027,083 ns
Selection Sort:  [1121.8739, 1131.5066, 1135.9407, 1136.3994, 1137.011, 1137.4697, 1141.4451, 1146.7966, 1149.3959, 1163.4627, 1241.565, 1242.26, 1242.816, 1252.407, 1261.442, 1261.859, 1263.249, 1391.5287, 1515.3449, 1526.312,
-------- 2 Threads running 'Insertion Sort' sorting function -------
Execution Time: 58,728,958 ns
Insertion Sort:  [1121.8739, 1131.5066, 1135.9407, 1136.3994, 1137.011, 1137.4697, 1141.4451, 1146.7966, 1149.3959, 1163.4627, 1241.565, 1242.26, 1242.816, 1252.407, 1261.442, 1261.859, 1263.249, 1391.5287, 1515.3449, 1526.312,
-------- 4 Threads running 'Insertion Sort' sorting function -------
Execution Time: 34,783,250 ns
Insertion Sort:  [1121.8739, 1131.5066, 1135.9407, 1136.3994, 1137.011, 1137.4697, 1141.4451, 1146.7966, 1149.3959, 1163.4627, 1241.565, 1242.26, 1242.816, 1252.407, 1261.442, 1261.859, 1263.249, 1391.5287, 1515.3449, 1526.312,
-------- 8 Threads running 'Insertion Sort' sorting function -------
Execution Time: 17,500,208 ns
Insertion Sort:  [1121.8739, 1131.5066, 1135.9407, 1136.3994, 1137.011, 1137.4697, 1141.4451, 1146.7966, 1149.3959, 1163.4627, 1241.565, 1242.26, 1242.816, 1252.407, 1261.442, 1261.859, 1263.249, 1391.5287, 1515.3449, 1526.312,
-------- 16 Threads running 'Insertion Sort' sorting function -------
Execution Time: 13,205,416 ns
Insertion Sort:  [1121.8739, 1131.5066, 1135.9407, 1136.3994, 1137.011, 1137.4697, 1141.4451, 1146.7966, 1149.3959, 1163.4627, 1241.565, 1242.26, 1242.816, 1252.407, 1261.442, 1261.859, 1263.249, 1391.5287, 1515.3449, 1526.312,

Process finished with exit code 0
```

Figure 3: Results for Selection Sort and Insertion Sort Running on 2, 4, 8, and 16 Threads



Search Algorithm: Linear

Number of Threads: 4

45

☑ age

□ sex

□ bmi

□ smoker

□ region

□ children

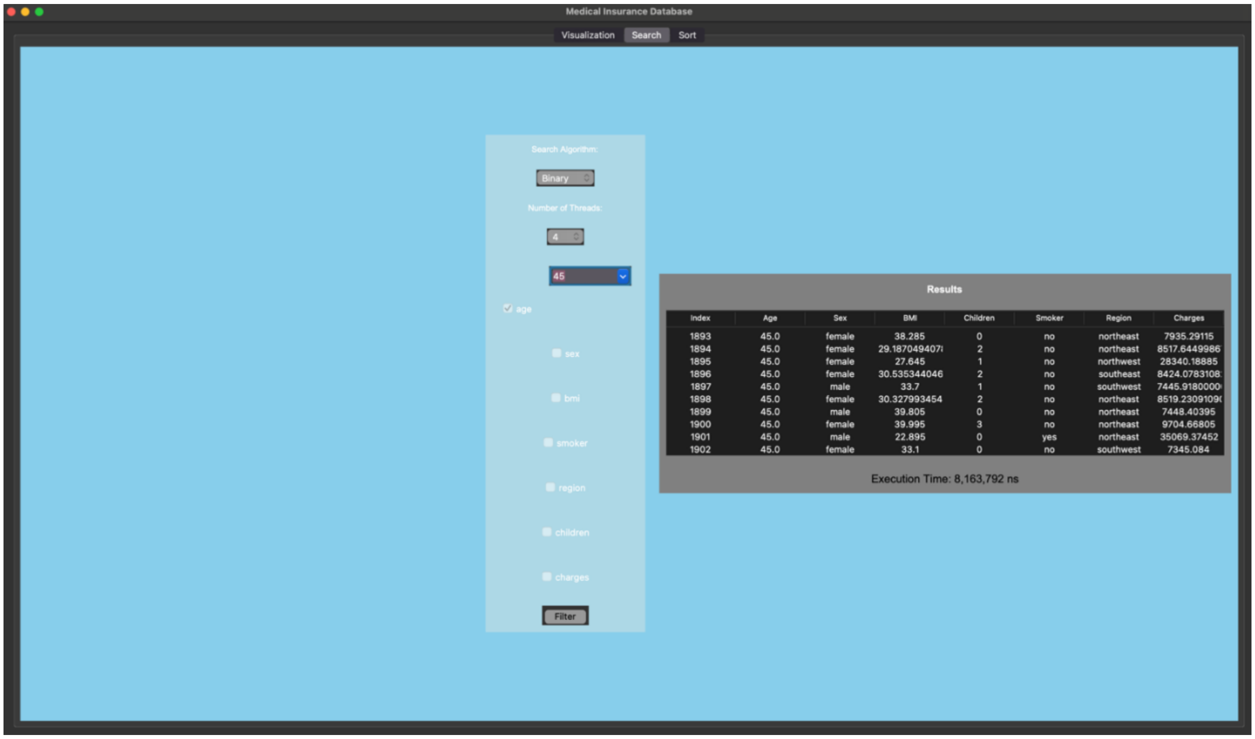
□ charges

Results

Index	Age	Sex	BMI	Children	Smoker	Region	Charges
11	45.0	female	26.9174112730	2	no	northeast	8549.1371546
68	45.0	female	30.327993454	2	no	northeast	8519.2309109
114	45.520535235	male	25.692120428	5	no	southeast	10019.490257
181	45.329935650	female	24.8016041197	5	no	southeast	9890.5204267
210	45.2318623131	female	26.389929743	5	no	southeast	9962.6123640
212	45.090232979	male	25.618339153	3	no	northeast	9137.7806609
273	45.733471364	female	28.4265912177	5	no	southwest	13145.568714
289	45.168480600	male	37.434094878	3	no	northwest	9670.2820019
319	45.748820907	male	29.404720046	5	no	southeast	10711.7887748
379	45.040810453	female	25.710475642	5	no	southwest	11497.8216189

Execution Time: 46,328,542 ns

Figure 4: Results for Linear Searching on Age = 45 with 4 threads



Search Algorithm: Binary

Number of Threads: 4

45

☑ age

□ sex

□ bmi

□ smoker

□ region

□ children

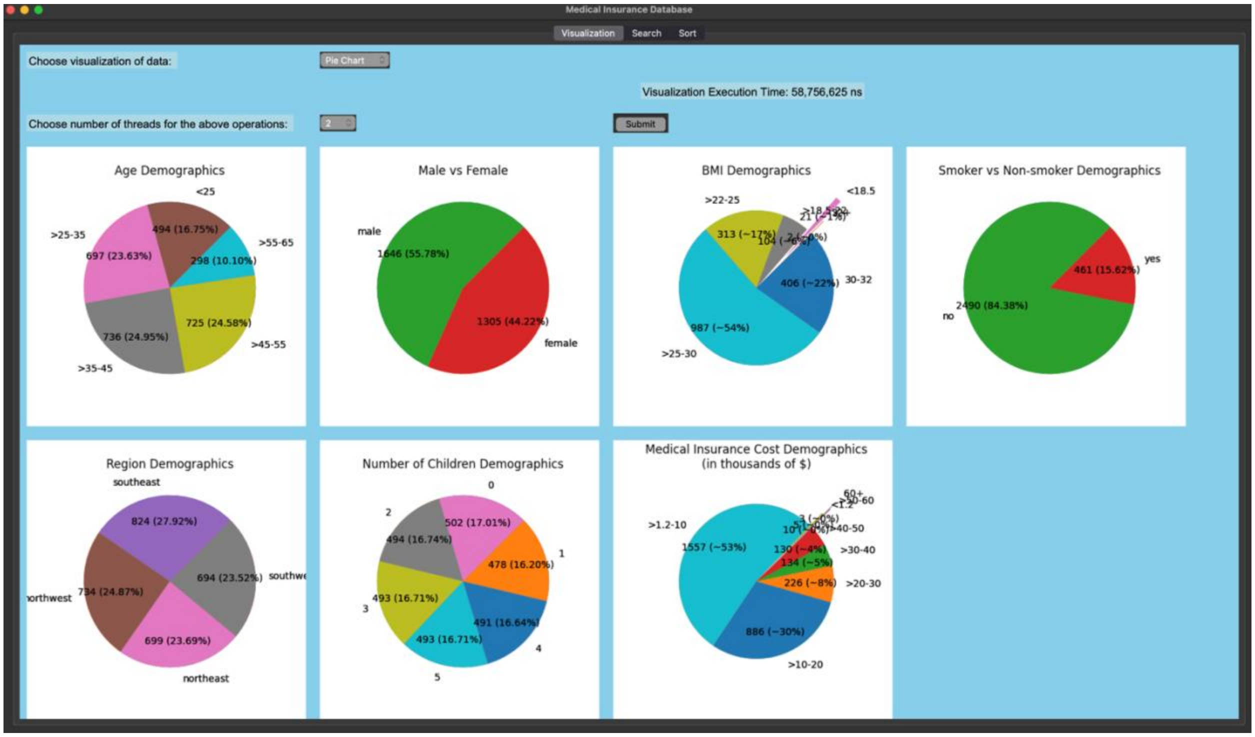
□ charges

Results

Index	Age	Sex	BMI	Children	Smoker	Region	Charges
1893	45.0	female	38.285	0	no	northeast	7935.29115
1894	45.0	female	29.187049407	2	no	northeast	8517.6449986
1895	45.0	female	27.645	1	no	northwest	28340.18885
1896	45.0	female	30.535344046	2	no	southeast	8424.0783108
1897	45.0	male	33.7	1	no	southwest	7445.9180000
1898	45.0	female	30.327993454	2	no	northeast	8519.2309109
1899	45.0	male	39.805	0	no	northeast	7448.40395
1900	45.0	female	39.995	3	no	northeast	9704.66805
1901	45.0	male	22.895	0	yes	northeast	35069.37452
1902	45.0	female	33.1	0	no	southwest	7345.084

Execution Time: 8,163,792 ns

Figure 5: Results for Binary Searching on Age = 45 with 4 threads

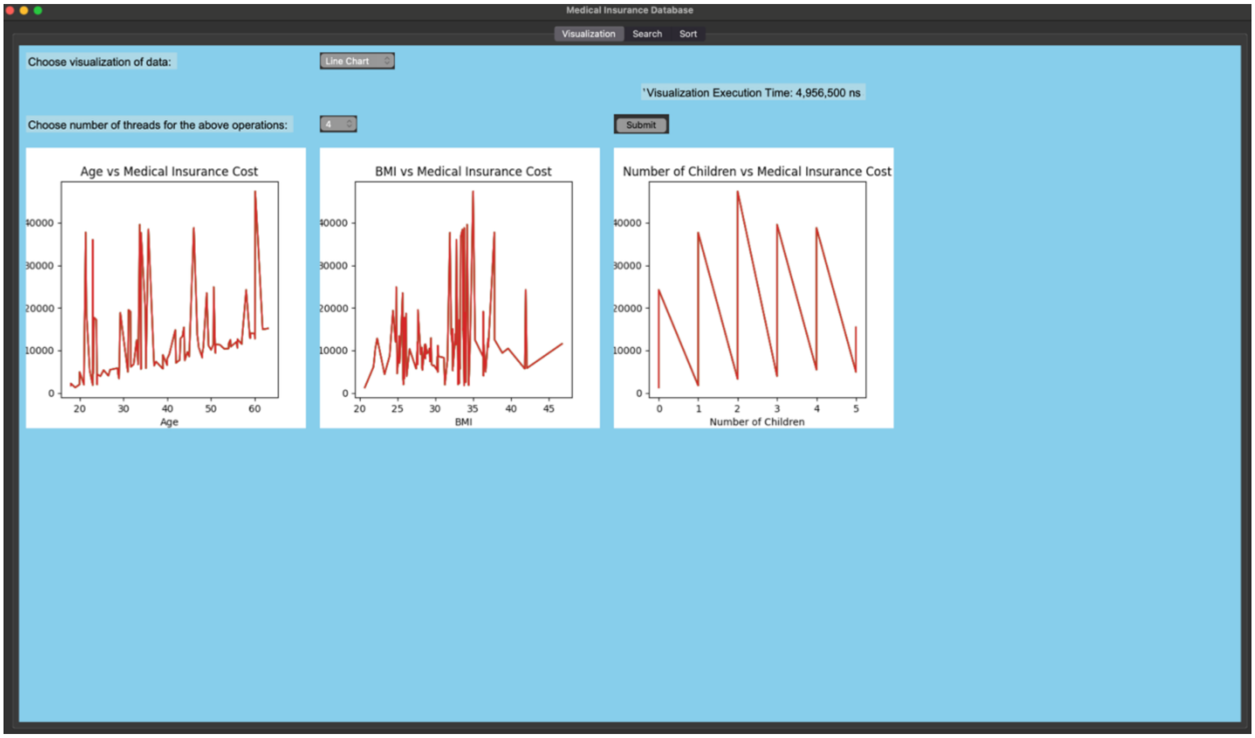


Choose visualization of data: Pie Chart

Visualization Execution Time: 58,756,625 ns

Choose number of threads for the above operations: 2

Figure 6: Results for Pie Chart Visualization on 2 Threads

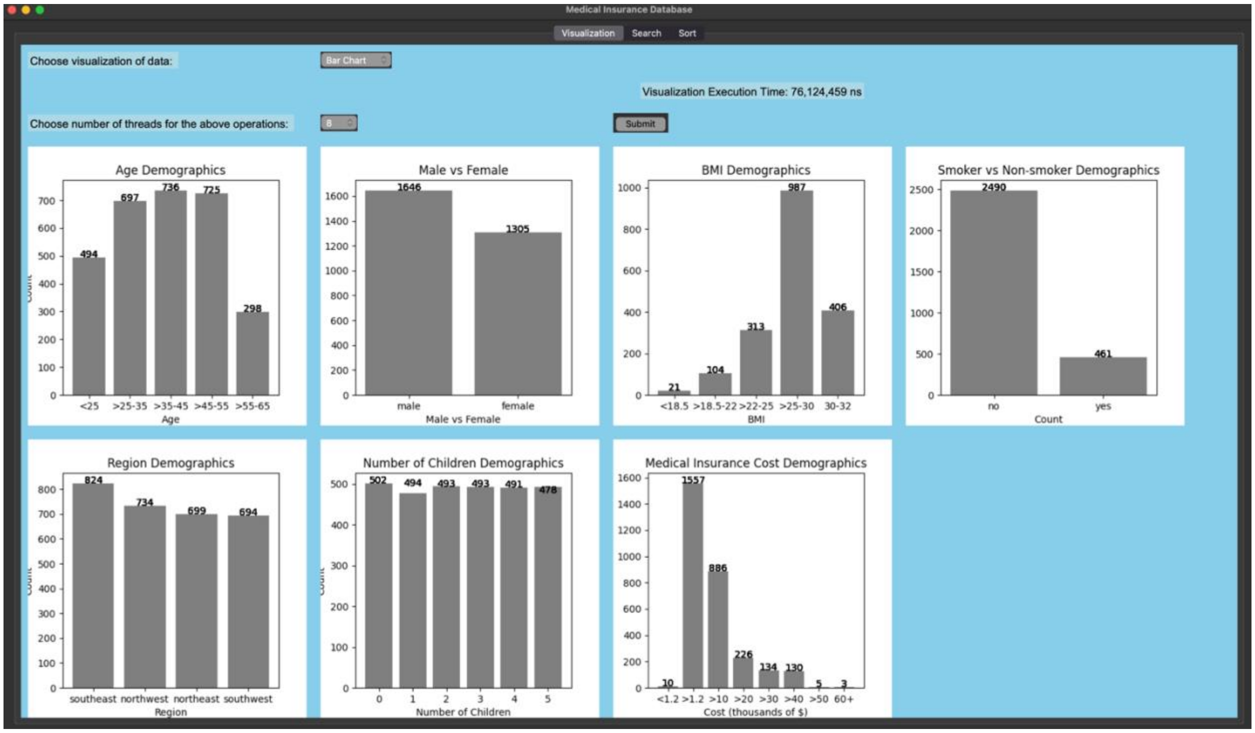

Medical Insurance Database
Visualization
Search
Sort
Submit
Age vs Medical Insurance Cost
0
10000
20000
30000
40000
20
30
40
50
60
Age
BMI vs Medical Insurance Cost
20
25
30
35
40
45
BMI
Number of Children vs Medical Insurance Cost
0
1
2
3
4
5
Number of Children

Choose visualization of data: Line Chart

'Visualization Execution Time: 4,956,500 ns

Choose number of threads for the above operations: 4

Figure 7: Results for Line Chart Visualization on 4 Threads



Choose visualization of data: Bar Chart

Visualization Execution Time: 76,124,459 ns

Choose number of threads for the above operations: 8

Figure 8: Results for Bar Chart Visualization on 8 Threads

Our demonstration video is linked here:

https://psu.mediaspace.kaltura.com/media/CMPSC+472+Project+2+-+Demo+Video/1_e530fhn8

Functionalities

In our application, we have three core functionalities for our medical insurance database simulation system. Firstly, we have sorting. Our sorting process allows the user to sort by selection sort (Halim, 2011) or insertion sort (DSA Insertion Sort, W3Schools). From these two algorithms, an execution time is provided, showing the user how long it takes to sort the data by medical insurance charge.

Secondly, we have a searching functionality that utilizes a linear and binary search approaches. From here, the user can query the data by filtering through the data based on specific values. For example, the data can be filtered by all patients of age 45, or the data can be filtered by any of the other column attributes in our medical insurance dataset. From here, similarly to the sorting functionality, the execution time is shown along with a visualization of the data frame of the specific filter data pulled from the medical insurance dataset.

Initially, in our sorting procedure, due to threads using separate data chunks of the column they are sorting, we considered implementing a function like the merge helper function that is used in a merge sort algorithm. However, after using 2 threads, we found that merging more than 2 lists at a time was not memory or time efficient, as many merged list objects were created (Kartik, 2025a). To improve upon this, we found a function from the heapq library (Google Search AI Overview), which provides a sorted list merging function for an n number of iterables (Manjeet, 2025; Mishra, 2025). This made it far more efficient to combine the data chunks form large thread amounts (such as 8 or 16) after they were sorted.

Thirdly, we included the functionality of visualization, the users can view all the medical insurance dataset as multiple pie charts, bar charts, or line charts using Matplotlib (Gavande, 2025). Again, an execution time is provided along with these charts. A limitation we found is that Tkinter and Matplotlib are traditionally not thread safe, but there is a workaround. Managing Tkinter and Matplotlib visuals in the main thread and isolating our other operations in other threads, returning only their execution results creates a thread safe environment. Therefore, for visualizations, we included many locks on the functions to ensure

Within all three of these core functionalities, each of the functionalities is backed by multi-threading. The user can choose how many threads (2, 4, 8, or 16) to use for a given function call. Our execution time result is the key feature of our program, exemplifying the importance of threading within large system and showing varying execution time results based on the number of threads used for a given function call.

Execution Results and Analysis

In our sorting process, we received expected results. Selection sort and insertion sort performed similarly as they both share a time complexity of O(n^2) in the worst case. However, using multi-threading and increasing the thread count, we found the execution time decrease by using sections (data chunks) of the medical insurance charges column converted to a list.

In our search process, we acknowledged that Linear search has a time complexity of O(n) while binary search has an average case of O(log n). Logically, binary search was faster in most cases. We decided to include both searching algorithms to fully analyze how multithreading can affect performance and reduce execution time.

In our visualization process, we received unexpected results. We expected more threads to always conclude that there will be a lower, improved execution time. However, we found the opposite. According to Google search AI Overview and Stack Overflow, we found that due to the intensive nature of the process of plotting, this increases the contact switching thus incurring a higher execution time rather than lower.

In our searching process, we received unpredictable results. Linear searching has a time complexity of O(n) in the worst case, and binary search has a time complexity of O(logn) in the worst case. Increasing or decreasing the thread count on these operations did not change the execution time to an accurate conclusion. In some of our tests, linear/binary search would improve with more threads, and in other tests, it would yield a higher execution time. We did consistently see improved execution results in binary search over linear search, but there were occasional trials of higher execution time.

Conclusion

In our project, we demonstrated the importance of multithreading on complex operations with large amounts of data. We simulated the use of a medical insurance company's database system and the process of querying/searching, sorting, and visualizing the data. We displayed the execution time of our core functionalities to demonstrate the importance of multithreading in a large system. We received some results that were unexpected, and we quickly learned that multithreading does not directly equate to an improved execution time; rather, it is important to consider overhead and how multithreading can affect the execution time both by improving or increasing the time. Additionally, it is important to consider the operation at hand, the data being used, and how the time complexity can be affected. A higher time complexity in the worst case does not always equate to improvement with multithreading.

Overall, we learned the importance of multithreading, but we also learned its limitations and the affects it can have on a system. We also learned about the importance of thread isolation within a system using different libraries that may not be inherently thread safe by default. We adapted workarounds to unsafe libraries to threads to execute our results by isolating their testing.

References

Adeem, M. (2025, July 15). *place_info(), pack_info(), and grid_info() methods in Tkinter.* GeeksforGeeks. https://www.geeksforgeeks.org/python/place_info-pack_info-and-grid_info-methods-in-tkinter/

Anderson, J. (n.d.). *Intro to Python Threading.* Real Python. https://realpython.com/intro-to-python-threading/

Badole, M. (2024, July 9). *Python Daemon Threads.* GeeksforGeeks. https://www.geeksforgeeks.org/python/python-daemon-threads/

Buffersnuff & Ghost (2019, February 2). How do I get my frame sizes to match up to my root window size? Stack Overflow. https://stackoverflow.com/questions/54498381/how-do-i-get-my-frame-sizes-to-match-up-to-my-root-window-size

Čongrády, M. (2015, July 4). *Python thread executing function twice.* Stack Overflow. https://stackoverflow.com/questions/31224845/python-thread-executing-function-twice

Doe, K. (2019, January 17). *How to make Tkinter GUI thread safe?* Stack Overflow. https://stackoverflow.com/questions/54237067/how-to-make-tkinter-gui-thread-safe

DSA Insertion Sort (n.d). W3Schools. https://www.w3schools.com/dsa/dsa_algo_insertionsort.php

Dutta, G. (n.d.). *Medical Insurance Cost*. Kaggle. https://www.kaggle.com/datasets/gauravduttakiit/medical-insurance-cost/data?select=Train_Data.csv

F, C. & Zwiker, D. (2015, February 15). *How to call a function on a running Python thread.* Stack Overflow. https://stackoverflow.com/questions/19033818/how-to-call-a-function-on-a-running-python-thread

Gavande, J. (2025, July 2025). *Bar Plot in Matplotlib.* GeeksforGeeks. https://www.geeksforgeeks.org/pandas/bar-plot-in-matplotlib/

Getting every child widget of a Tkinter window. (n.d.). Tutorials Point. https://www.tutorialspoint.com/getting-every-child-widget-of-a-tkinter-window

Google Search AI Overview: syntax/debugging questions for Python threading, Matplotlib, and Tkinter.

Grendel. (2024, April 25). *Adding labels within a pie chart in Python by optimising space*. Stack Overflow. https://stackoverflow.com/questions/78383395/adding-labels-within-a-pie-chart-in-python-by-optimising-space

Halim, S. et al. (2011). *Visualgo.Net/sorting.* Visualgo.Net. https://visualgo.net/en/sorting

hkml. (2019, September 2). *Split large Dataframe into smaller equal dataframes.* Stack Overflow. https://stackoverflow.com/questions/57753041/split-large-dataframe-into-smaller-equal-dataframes

Jain, R. (2025, April 23). *Linear search using Multi-threading.* GeeksforGeeks. https://www.geeksforgeeks.org/dsa/linear-search-using-multi-threading/

Kartik. (2025a, October 3). *Merge Sort.* GeeksforGeeks. https://www.geeksforgeeks.org/dsa/merge-sort/

Kartik. (2025b, October 3). *Multithreading in Python*. GeeksforGeeks. https://www.geeksforgeeks.org/python/multithreading-python-set-1/

Kartik. (2025c, November 6). *Binary Search | Python.* GeeksforGeeks. https://www.geeksforgeeks.org/python/python-program-for-binary-search/

Kartik. (2025d, December 6). *Insertion Sort Algorithm.* GeeksforGeeks. https://www.geeksforgeeks.org/dsa/insertion-sort-algorithm/

Khandelwal, Y. (2024, July 1). *time.perf_counter() function in Python.* GeeksforGeeks. https://www.geeksforgeeks.org/python/time-perf_counter-function-in-python/

Manjeet (2025, July 11). *Combining Two Sorted Lists – Python.* GeeksforGeeks. https://www.geeksforgeeks.org/python/python-combining-two-sorted-lists/

Mishra, S. (2023, March 4). *Merge two sorted arrays in Python using heapq*. GeeksforGeeks. https://www.geeksforgeeks.org/python/merge-two-sorted-arrays-python-using-heapq/

Notanewbiedude. (n.d.). *How does tkinter multithreading work, and why?* Reddit. https://www.reddit.com/r/learnpython/comments/10qzto6/how_does_tkinter_multithreading_work_and_why/

Pandas Merge. (n.d.). ProgramIz. https://www.programiz.com/python-programming/pandas/merge

Prernadec. (2025, July 23). *Line chart in Matplotlib – Python*. GeeksforGeeks. https://www.geeksforgeeks.org/python/line-chart-in-matplotlib-python/

Pie Charts (n.d.) Matplotlib. https://matplotlib.org/stable/gallery/pie_and_polar_charts/pie_features.html

Rani, B. (2025, August 29). *UML Class Diagram.* GeeksforGeeks. https://www.geeksforgeeks.org/system-design/unified-modeling-language-uml-class-diagrams/

Renganathan, K. (2018, May 25). *Why more number of threads takes more time to process?* Stack Overflow. https://stackoverflow.com/questions/50525849/why-more-number-of-threads-takes-more-time-to-process

Roseman, M. (2007-2025). The Grid Geometry Manager. TkDocs. https://tkdocs.com/tutorial/grid.html

Sharma, A. (2025, July 28). *Create First GUI Application using Python-Tkinter.* GeeksforGeeks. https://www.geeksforgeeks.org/python/create-first-gui-application-using-python-tkinter/

Shipman, J. (2013). *Tkinter 8.5 reference: a GUI for Python*. TkDocs. https://tkdocs.com/shipman/grid.html

Shivansh (2025, July 23). *Adding value labels on a Matplotlib Bar Chart.* GeeksforGeeks. https://www.geeksforgeeks.org/python/adding-value-labels-on-a-matplotlib-bar-chart/

Threading – Thread-based parallelism. (2025, December 8). Python Docs. https://docs.python.org/3/library/threading.html

Tkinter – Python interface to Tcl/Tk. (2025, Decemeber 8). Python Docs. https://docs.python.org/3/library/tkinter.html#threading-model

William, J. (2025, July 9). *Use Tkinter to Design GUI Layouts.* Python GUIs. https://www.pythonguis.com/tutorials/use-tkinter-to-design-gui-layout/