David Pevahouse Lab 4

There were a few assumptions made before the beginning of this lab that testing the data could either prove or disprove. Some of the things that I assumed would have happened did, and others did not. Looking at the different sorts, I assumed that the first item pivot quick sort would be the worst in terms of efficiency, on average, due to the high likelihood, especially of already sorted files, that it would lead to worst case situations that are highly inefficient. I had also assumed that the insertion sort of 50 would be more efficient then 100 but less efficient then the median of three pivot option. Finally, my last assumption, was that the merge sort would end up being the most efficient on average due to some of the large file sizes. In fact, in the course of this lab, the given data would be around 20,000. Relatively small for real world scenarios but not insignificant. One of the additional inputs added to the lab increased that size to 60,000 to further test, with numbers being nearly completely random and not necessarily limited to being smaller then 60,000.

Before I get into analyzing my findings against the data received, I wanted to describe a bit of what was done, as well as some of the struggles I found along the way. My data structures used in this lab were fairly simple. In the natural merge, a simple linked list implementation of only pointing to the next node was used, and in all of the quick sorts, several arrays were used to make sure to account for the breaking down of the data based on what was the pivot and how to further decrease the sample size of data to try and decrease the amount of comparisons. From the data structures perspective, I wanted to keep it simple to make sure to focus on seeing the levels of efficiency from the sorts. There are two function that is used on every file to add the data to an array and linked list.

That does lead me to the first issue I had with that. Some of the files were organized in a way where all of the data was on one line, and other files were organized where it was one piece of data per line. That made it a little tricky to assign the data to an array or linked list and for a while I was only putting in one number into the data structures. Not only did that originally make the data incredibly inaccurate, it also made the data increase to an incredible limit. In order to account for that, a simple workaround was implemented. Essentially, the algorithm looks that if the data in the line 1 before it is a number, it concatenates both of them together instead of creating two separate entries, making sure to account for multi-digit numbers. Another issue on the way of fixing this was that since all of the sorts was decided to be implemented using recursion, and the file sizes were rather large, an increase to the system recursion limit was required. At first 50,000 was tested, and in the end that was to small and 50,000 was selected and was verified to be functional. An exact number or science was not applied here, so given more time a greater efficient number would have been discovered and possibly more efficiency to the recursive algorithm's themselves could have been found. Funnily enough, another issue that I had was making sure that counting the exchanges and comparisons was kept at the right rate. I think this was more of a moment of struggling to connect the dots of how to do so.

I believe that the efficiency of these algorithms follow the general given. For both of these algorithms a space and time of O(nlogn) was required to perform these functions. That was accomplished by making sure that a new set of data was not created for every recursive algorithm, simply passing the data structure, or portions of it, from function to function to conserve as much as possible. Though, in theory, it is possible that the worst case scenario of these sorts would far exceed the standard of nlogn. For example, in a completely sorted list using the first pivot in a quick sort, there is not a lot of difference between that and a insertion sort, making the complexity approach n^2 or even exceed due to some of the extra calculations needed to do essentially the same job.

Some of the enhancements I added to the goal is for one, reading in all of the files at the same time, or being able to insert a directory instead of individual files, to make testing all of the data a breeze instead of having to go through them one by one. Another would be the amount of data the algorithm can take in. The lab

specifies that up to 10,000 integers are required, however one of my new input allows up to 60k to be inserted and to be sorted. An improvement that will add a good deal of value to this analysis will be the fact that the algorithm creates a csv instead of just a simple text file. That allows the use of excel, or other applications, to read in the data and create graphs that can make the data more human readable and easy to get a true feel of what was going on in the data. It was also decided to record the amount of time it took to sort each file, just to make another comparison on the efficiency of each algorithm on each file, also adding the name of the file that was sorted as well, just to make sure that the data all aligns. Another enhancement was that for the insertion sort algorithm, I made it easy to change the data, since we only used 100 and 50 that is all that is on example here, but another could easily be thrown in by changing the variable k in that algorithm.

Now to analyzing the data. Right of the back, something that caught me completely off guard and made me think something was wrong with the algorithms (there still might be) was the at the median of three quick sort was by far much, much shorter in time then all of the other sorting algorithms. My only guess is that the others, with the randomness of the data (or lack thereof) actually leads moreso to the worst case scenario of n^2 or worse, where the median of three assures that we stay close to nlogn or even better. By looking at the data below it is really just quite impressive. Even more so considering that the comparisons are not to far off. One flaw of the data is my extremely large input file does make the graphs slightly harder to read. In hindsight, it would have been a good idea not to make a file that large for that sake, but I did want to see how it affected all of the data.

There is a file, output.txt that will validate that the sorts are functioning as expected though, as well as showing how long it took for the algorithm to run through all of the data together (around 700 seconds on my computer). This shows that the data is getting sorted appropriately, which was needed considering the vast differences of the data. While the data does mostly support my hypothesis and assumptions I made in the first paragraph, I greatly did not expect the sheer amount of differences in the time it took to complete each one, mostly how all of them were relatively similar but the median of three was incredibly faster. The next fastest was the natural merge sort, and while way faster then the other quick sorts, was still nearly three times slower than the median of three. For some reason it was also a bit of a surprise that the comparisons were similar between each of the algorithms as well.

If there was one big takeaway that I am happy to be walking away from this lab with, is just the sheer importance of making sure to select the most efficient option that you can when it comes to data. In my line of work, every millisecond counts, where we often have to guarantee that all of our software works within 2-16 miliseconds. 3 of the 5 algorithms here did not even get close to being as efficient in saving that time resource, and it is amazing to me the difference it made. Interestingly enough though, the difference in time between the natural merge sort and median of three, while definitely significant, did not seem to be overall to great when it came to incredibly large data sets. I am assuming that if I had a data set in the millions, it would be possible for the natural merge to even overtake the quick sort based on what the data was showing below. I am still curious if there was something I messed up along the way to make such a great difference between the quicksorts, and that is definitely something I would like to try in the future.

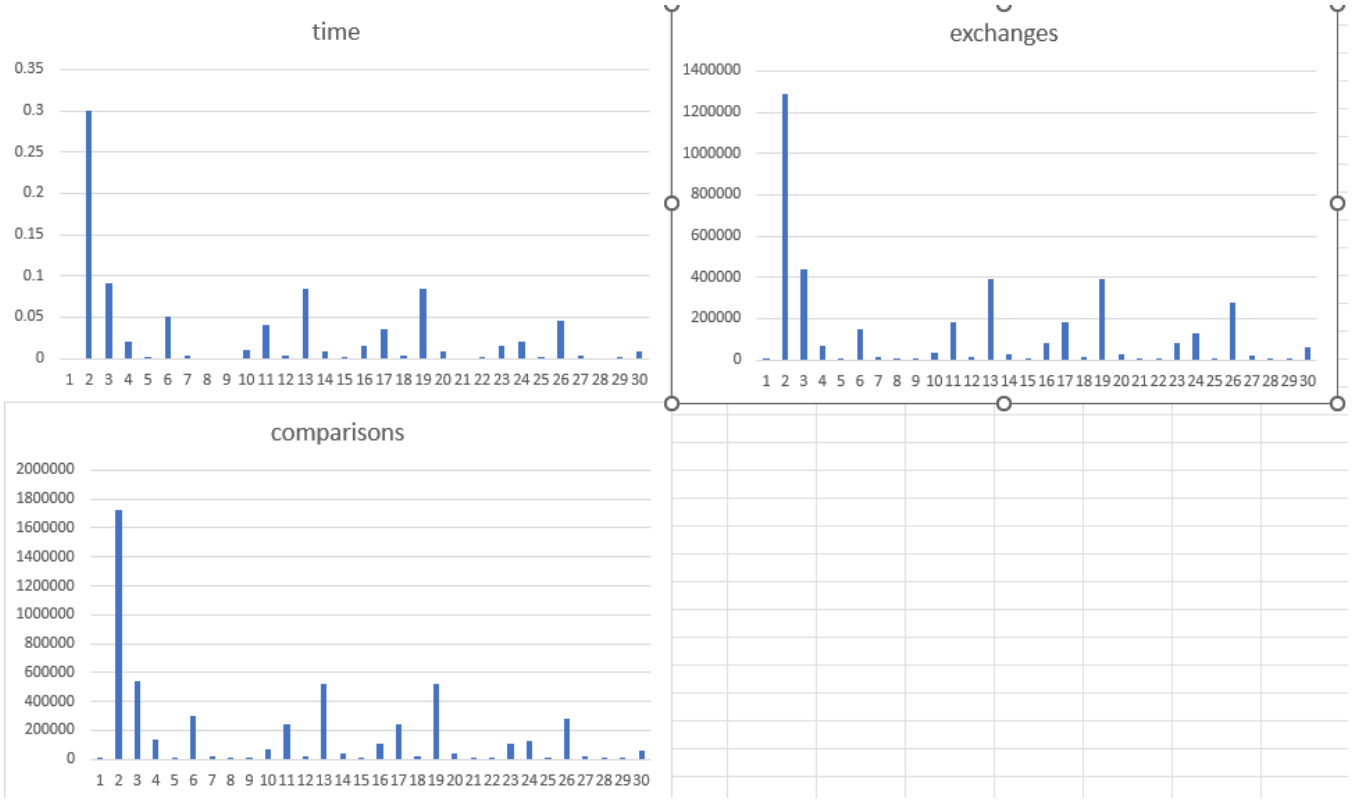
All in all, this lab was very informative and incredibly interesting. I enjoyed the fact that instead of focusing on trying to make something just work, I was moreso focusing on why certain algorithms would be chosen over others and how to take that into account. Not only that, but instead of just seeing theory of why that would be done, I got to see very real answers that showed just how much difference that can make. Even with a relatively, with real world standards, small dataset of 60k, we can see a difference in some of the lesser efficient algorithms of 30 seconds vs .12 seconds, which I do not even understand how that math works out it is so crazy to me. I enjoyed working and learning about this lab, and a copy of what my pycharm configuration file

is also attached below all of the graphs. As this is the last lab, thank you for all of the wonderful learning experiences and challenges that these brought me and let me know if you have any questions!

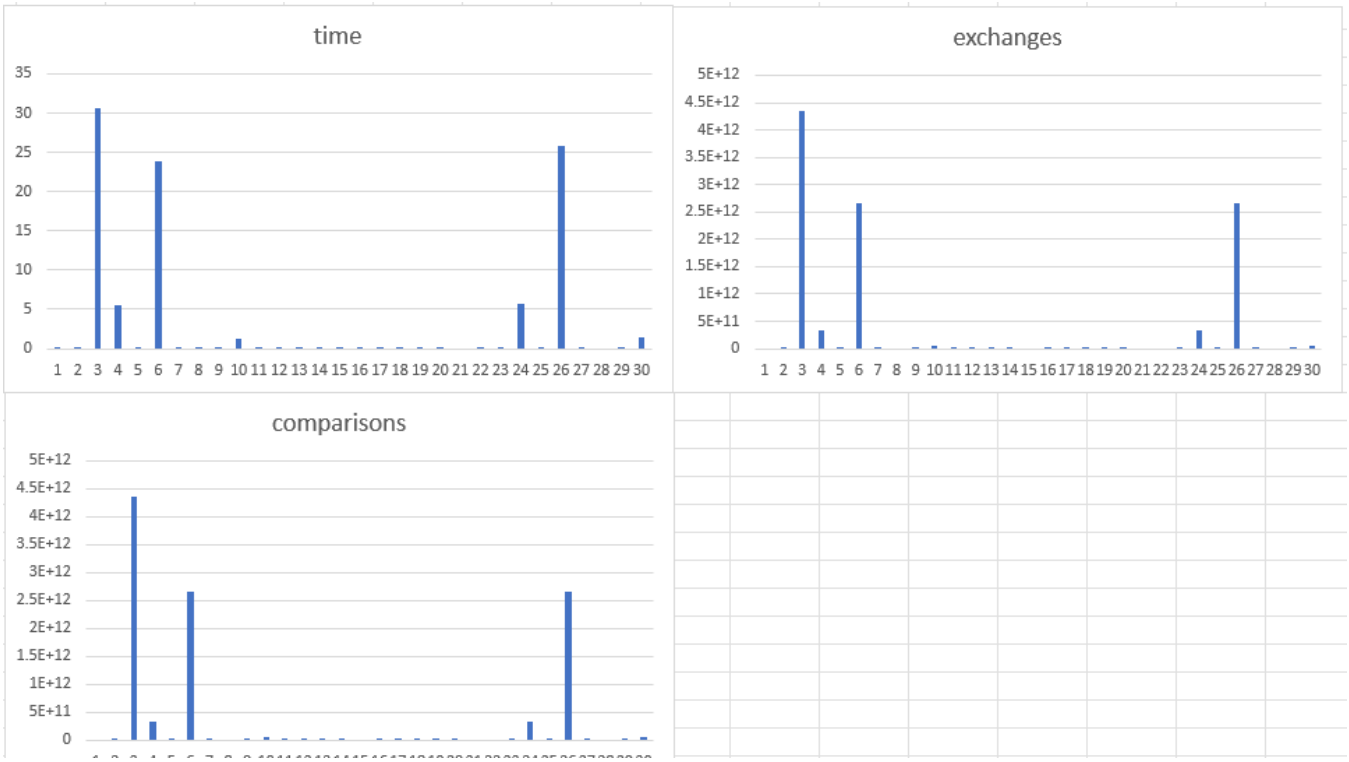
This is also found in the readme describing the output.csv and output.txt for what the variables mean just incase it was not clear right off the back.

```
ll = the natural merge sort
qs = the quick sort using first pivot
qsi100 = the quick sort using first pivot and insertion sort for n < 100
qsi50 = the quick sort using first pivot and insertion sort for n < 50
qsm = the quick sort using median pivot
```

Natural Merge Graphs

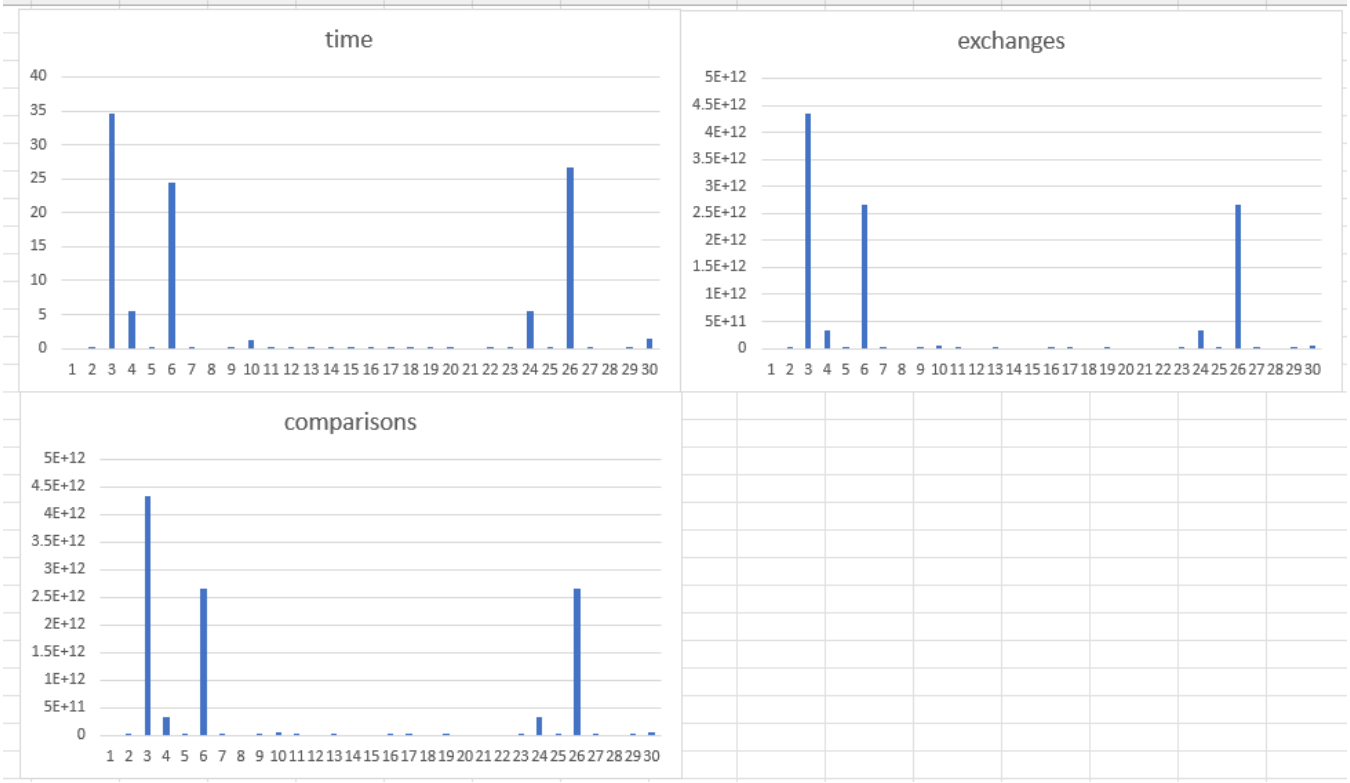


Quick Sort first pivot graphs

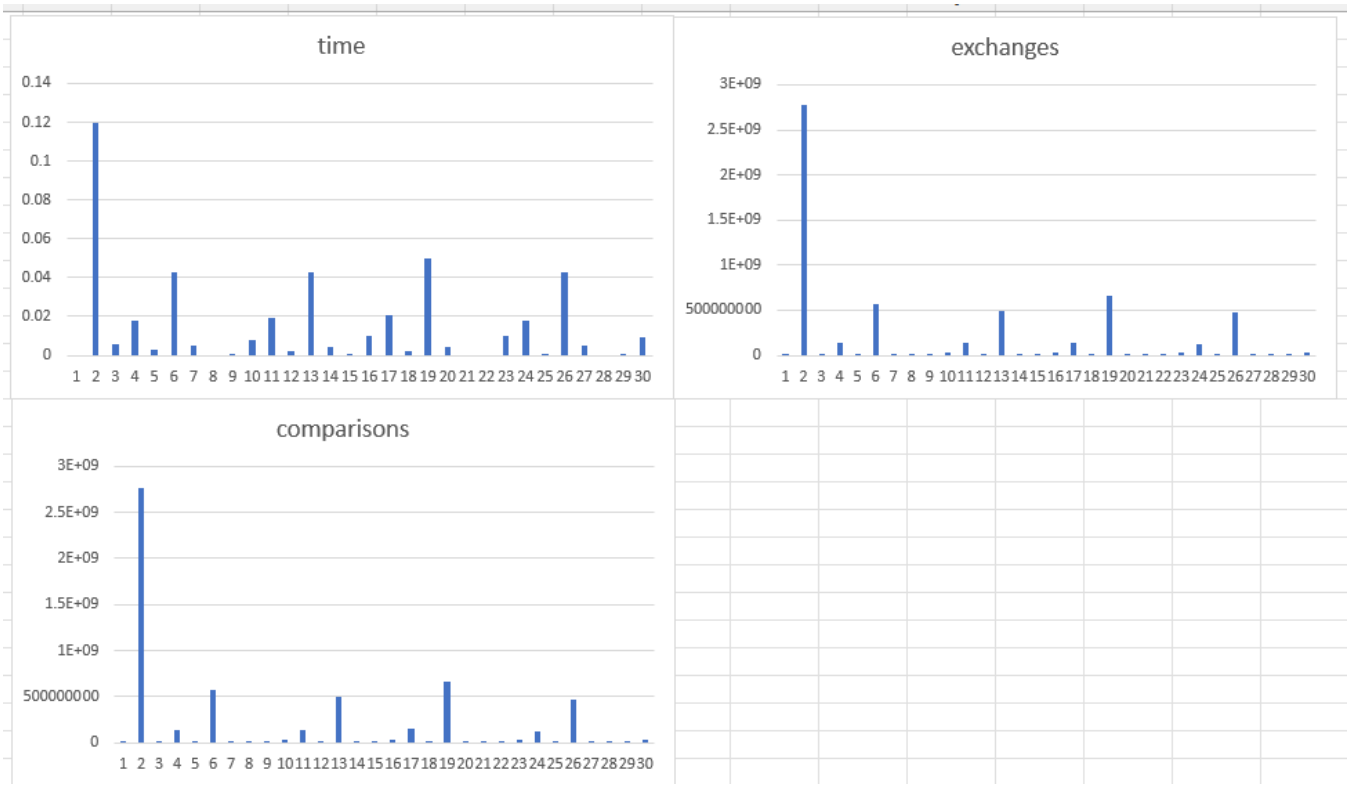


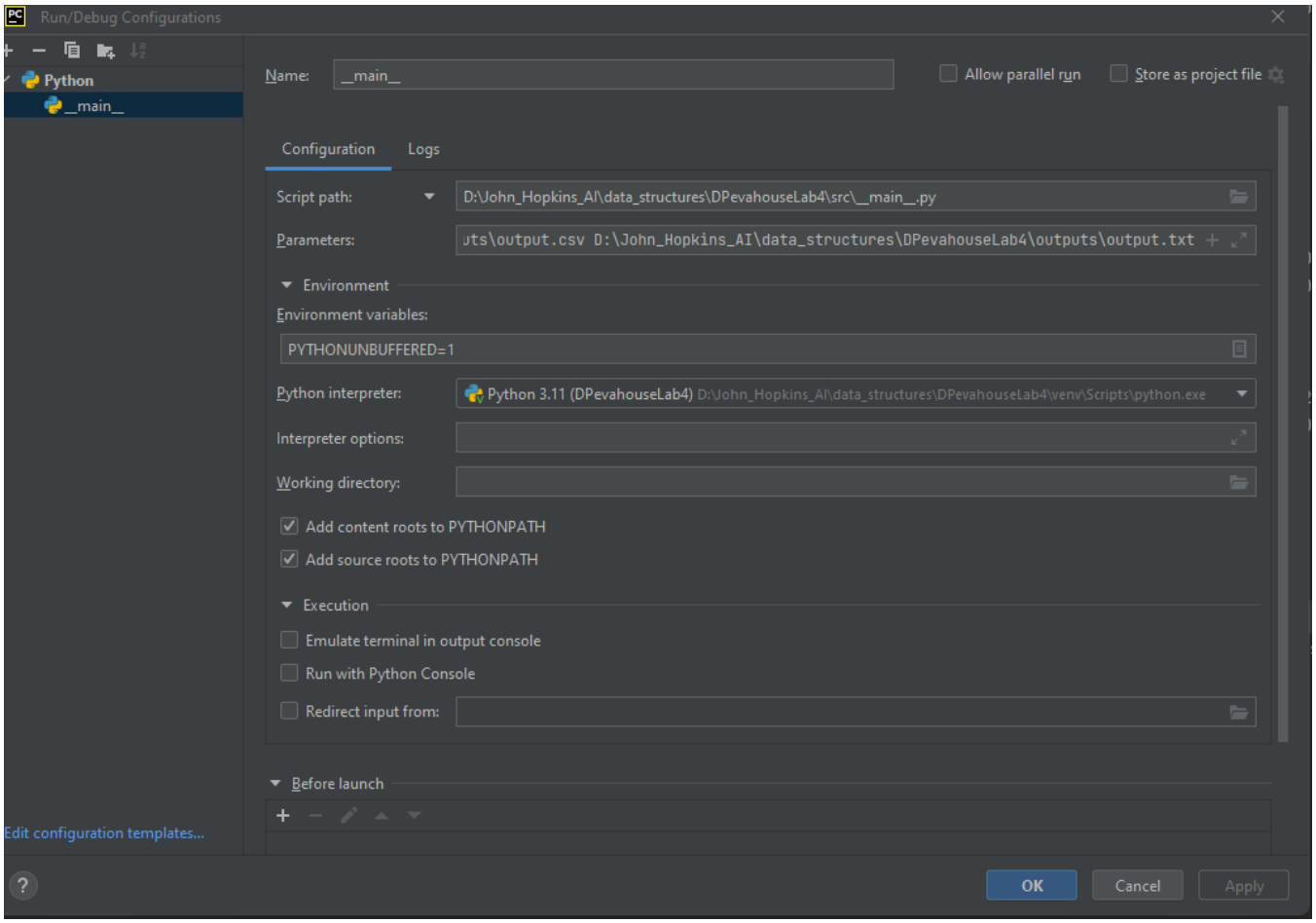
Quick sort 100 insertion

Quick Sort 50 insertion



Quick Sort median of 3


time
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
exchanges
3E+09
2.5E+09
2E+09
1.5E+09
1E+09
500000000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
comparisons
3E+09
2.5E+09
2E+09
1.5E+09
1E+09
500000000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30


Run/Debug Configurations
Python
__main__
Name: __main__
Allow parallel run
Store as project file
Configuration
Logs
Script path: D:\John_Hopkins_AI\data_structures\DPevahouseLab4\src__main__.py
Parameters: uts\output.csv D:\John_Hopkins_AI\data_structures\DPevahouseLab4\outputs\output.txt
Environment
Environment variables:
PYTHONUNBUFFERED=1
Python interpreter: Python 3.11 (DPevahouseLab4) D:\John_Hopkins_AI\data_structures\DPevahouseLab4\venv\Scripts\python.exe
Interpreter options:
Working directory:
Add content roots to PYTHONPATH
Add source roots to PYTHONPATH
Execution
Emulate terminal in output console
Run with Python Console
Redirect input from:
Before launch
Edit configuration templates...
OK
Cancel
Apply