Submission for	Final project report
Subject	Algorithm and Programming
Name of Lecturer	Jude Joseph Lamug Martinez, MCS
Made by	Tiffany Widjaja 2802503791
Location	BINUS UNIVERSITY INTERNATIONAL, JAKARTA, 2024
Project title	Big-O time complexity Visualizer

Table of Contents

Project Specifications

Project description

For my semester final project, I made a true-to-computer time complexity visualizer/diagram out of curiosity on whether some algorithms are true to theory.

While big O diagrams exist out there, it doesn't support custom algorithms. You can edit the algorithm in `algorithm_runner.py` to see the time complexity of your algorithm.

Since it was hard to see the millisecond time difference, I added more ways to view the diagram: by changing the settings or by using a different visualizer.

Project link

https://github.com/frost-drago/Big-O-Visualizer

How to use

Standard usage:

1. Run `pure_runner.py` to get the time complexity results of pure algorithms in your computer.
2. Edit `algorithm_runner.py` and run it to get the time complexity results of your custom algorithm in your computer.
3. Choose a visualizer:
    i. Matplotlib visualizer (run `visualizer_matplotlib.py`)
    ii. Plotly visualizer (run `visualizer_plotly.py`, dump the terminal after you've finished and want to run this again to avoid errors. The library is not robust. Keep rerunning and dumping the terminal because Plotly accesses random ports, some of which are not available.)

"I don't want to recalculate the whole thing, just my algorithm"

1. Follow steps in **[Standard usage]**, starting from no. 2

Custom colors, number of data to test, time limit, or number of trials to get average

1. Edit the config in `modules\settings.py`.
2. Edit `pure_runner.py` and `algorithm_runner.py` if necessary.
3. Follow steps in **[Standard usage]** again.

Python library dependencies

- `msgspec`
- `matplotlib`
- `plotly`

Program design

Runnables (are separate)

- `pure_runner.py` Calculate the time for well-known (pure) time complexity algorithms
- `algorithm_runner.py` Calculate the time for your custom algorithm
- `visualizer_matplotlib.py` Display the findings in a graph
- `visualizer_plotly.py` Display the findings in a graph

Modules

Module	Usage	Includes
`commons.py`	Functions that can be reused.	Function `get_list_until_number(a_list, target_number)`
		Function `average_of_a_list(a_list)`
		Function `random_list_generator(list_length, min_int=0, max_int=10000)`
		Function `random_list_generator_nonrepeating(list_length)`
		Function `num_of_data_range_generator(max_number_of_data)`
`settings.py`	Configuration (settings) class and instances of it.	Class `config`
`pure_algorithms.py`	Functions to run and track the time of pure time complexity algorithms.	Function $O(n!)$ `pure_n_factorial(a_list)`
		Function $O(2^n)$ `pure_2_to_the_power_of_n(a_list)`
		Function $O(n^2)$ `pure_n_to_the_power_of_2(a_list)`
		Function $O(n \log(n))$ `pure_n_times_log_n(a_list)`
		Function $O(n)$ `pure_n(a_list)`
		Function $O(\log(n))$ `pure_log_n(a_list)`
		Function $O(1)$ `pure_1(a_list)`
`sorting_algorithms.py`	Functions to run sorting algorithms.	Function `bubble_sort(a_list)`
		Function `selection_sort(a_list)`
		Function `insertion_sort(a_list)`
		Function `merge_sort(a_list)`
		Function `quick_sort(a_list)`
		Function `heap_sort(a_list)`

Algorithms

Algorithms included in the `pure_runner.py` :

- For `pure_n_factorial(a_list)` I decided to use an algorithm to generate permutations.
- For `pure_2_to_the_power_of_n(a_list)` I decided to use an algorithm to generate subsets.
- For `pure_n_to_the_power_of_2(a_list)` I decided to a nested loop.
- For `pure_n_times_log_n(a_list)` I decided to use merge sort. This is (almost) exactly the same code as the merge sort algorithm in `modules/sorting_algorithms.py`.
- For `pure_n(a_list)` I decided to use a plain loop.
- For `pure_log_n(a_list)` I decided to use an iterative logarithm algorithm. The list is not used, except for getting the length. The number is halved each iteration in a loop.
- For `pure_1(a_list)` I decided to just use `pass`. Though, somehow, the time very slightly grows. My hypothesis is that cause more system resources (processing power, memory) is being used for longer lists of random numbers. In theory, O(1) is supposed to remain constant.

Additional explanations of functions included in the `commons.py` :

- `get_list_until_number(a_list, target_number)` generates a sublist of a list from the beginning until an element is found. It is used in the process of generating `data/your_algorithm.json` which stores your custom algorithm's results.
- `num_of_data_range_generator(max_number_of_data)` generates a list of number of data to test. The beginning of the list starts out as dense, and eventually gets sparser.
- You can use `random_list_generator` or `random_list_generator` or the `range` function to help you generate data in `algorithm_runner.py` .

Other notes on algorithm design choices:

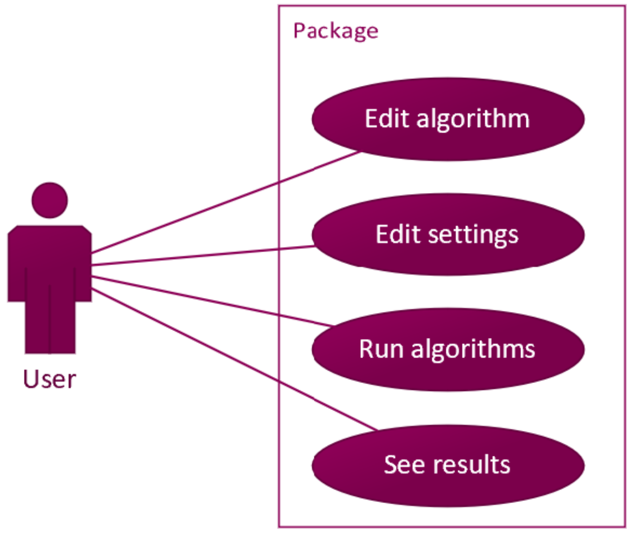
- All algorithms intentionally cut off (stop working) when their average time taken for a number of data exceeds `time_limit` (can be configured in `modules/settings.py`). I don't want a O(n!) algorithm calculate 4000 data, it will take a very, very long time.

UML Class diagram

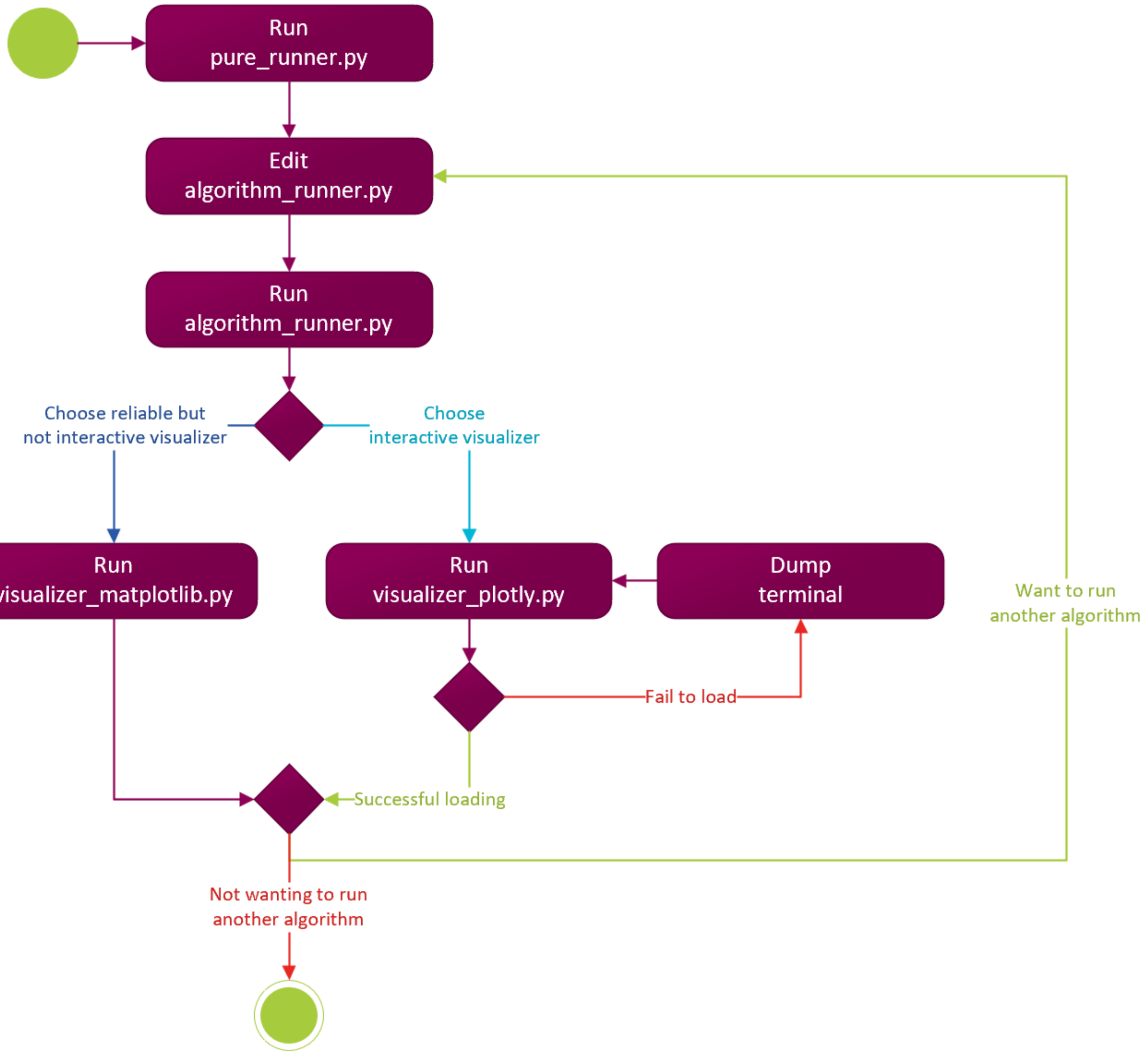
The class is made because it's mandatory to have at least one class with methods.

config
time_limit : int or float # max_num_of_data: int # colors : list # number_of_trials : int
- __init__(self, time_limit, max_num_of_data, colors, number_of_trials) + get_time_limit : int or float + get_max_num_of_data : int + get_colors : list + get_number_of_trials : int + set_time_limit(self, time_limit) + set_max_num_of_data(self, max_num_of_data) + set_colorsset_colors(self, colors) + set_number_of_trials(self, number_of_trials)

UML Use case diagram



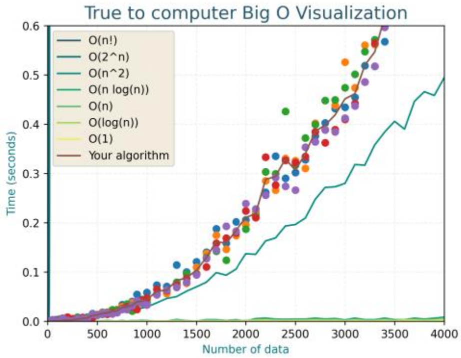
UML Activity diagram



Findings

<table>
<tr><th>Results</th><th>Comments</th></tr>
<tr><td>

Bubble sort



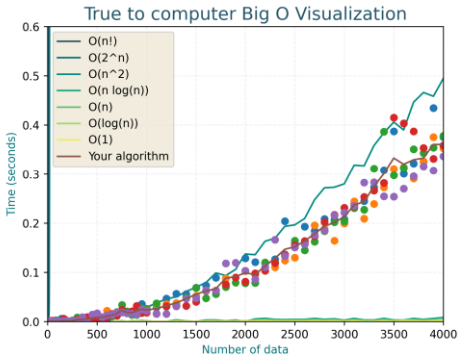
</td><td>

It uses `modules/sorting_algorithms.py`'s `bubble_sort()` function.

Results are **closest to O(n^2)** time complexity, though slower.

</td></tr>
<tr><td>

Selection sort



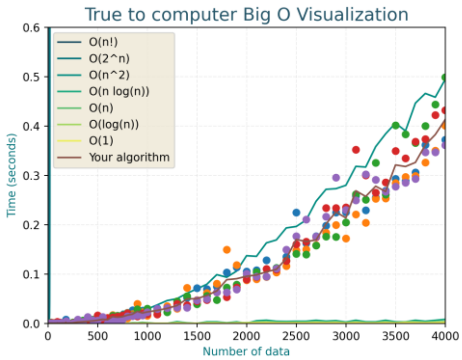
</td><td>

It uses `modules/sorting_algorithms.py`'s `selection_sort()` function.

Results are **closest to O(n^2)** time complexity, though faster.

</td></tr>
<tr><td>

Insertion sort



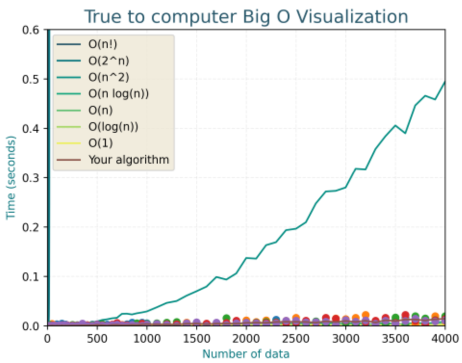
</td><td>

It uses `modules/sorting_algorithms.py`'s `insertion_sort()` function.

Results are **closest to O(n^2)** time complexity, though faster.

</td></tr>
<tr><td>

Merge sort

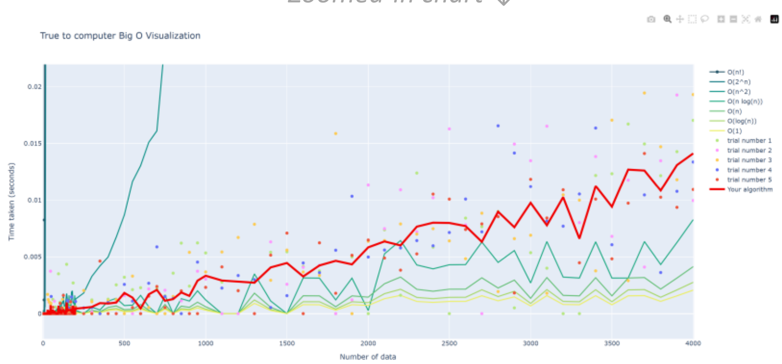


Continued ↓

</td><td>

It uses `modules/sorting_algorithms.py`'s `merge_sort()` function.

Zoomed-in chart ↓



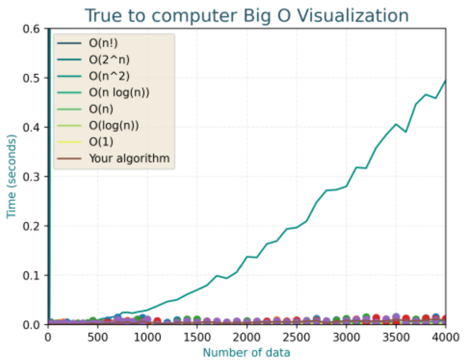
Results are **closest to O(n log(n))** time complexity, though a slower. I wonder why. The algorithms of `modules/pure_algorithms.py`'s `pure_n_times_log_n()`

</td></tr>
</table>

… Continued from above

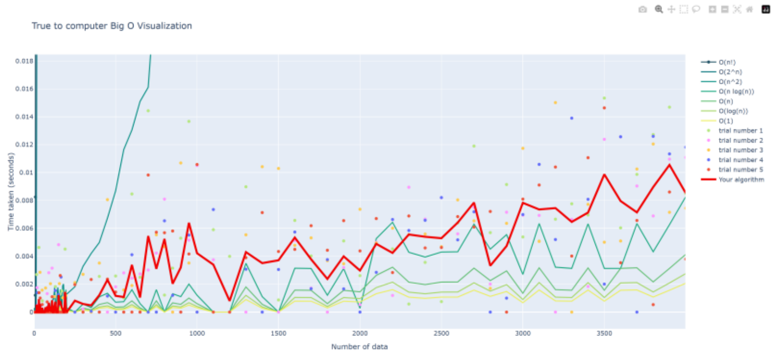
and `modules/sorting_algorithms.py`'s `merge_sort()` are exactly the same, so why the millisecond difference?

Quick sort



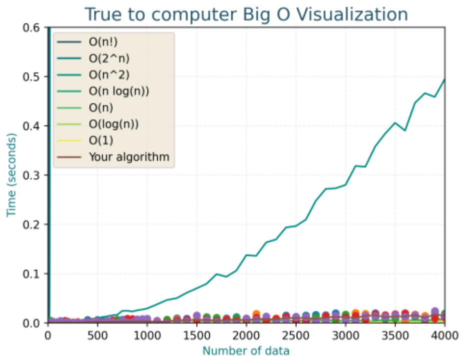
It uses `modules/sorting_algorithms.py`'s `quick_sort()` function.

Zoomed-in chart ↓



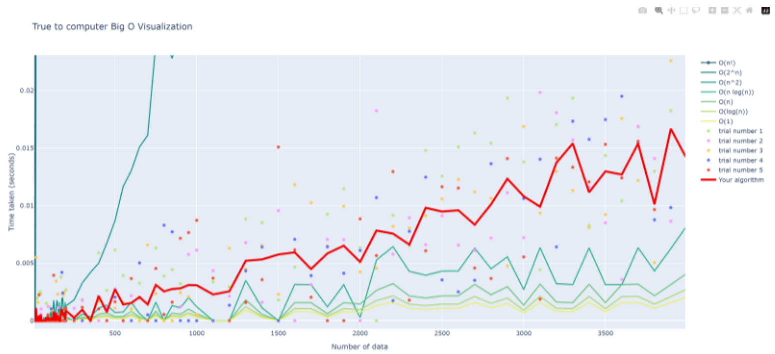
Results are **closest to O(n log(n))** time complexity, about the same.

Heap sort



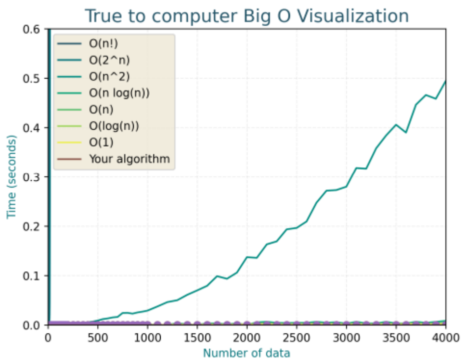
It uses `modules/sorting_algorithms.py`'s `heap_sort()` function.

Zoomed-in chart ↓



Results are **closest to O(n log(n))** time complexity, though slower.

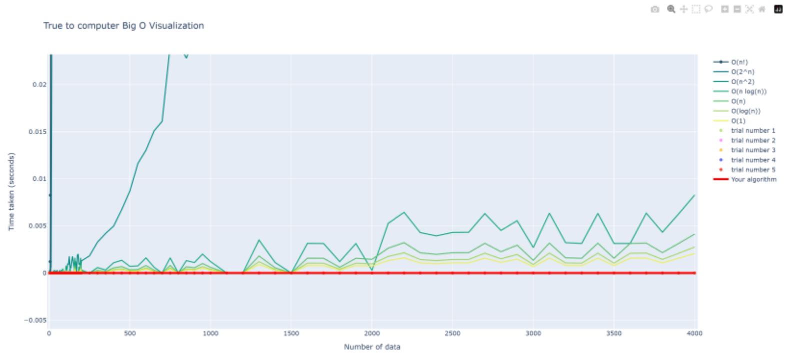
Pass (not a sorting algorithm)



`pass` is added to the algorithm section in `algorithm_runner.py`

```
########## EDITABLE 📝📝 ##########
########## Editable start ##########

# generate something (for example, an u
data_to_process = commons.random_list_ge

# track time (DO NOT EDIT)
time_start = time.time()

# run an algorithm (for example, a sorti
pass

########## EDITABLE 📝📝 ##########
########## Editable end    ##########
```

Continued ↓

<table>
<tr><th>… Continued from above</th><th>Zoomed-in chart ↓</th></tr>
<tr><td></td><td>


Results are closest to O(1) time complexity, though faster.
It’s because <code>modules/pure_algorithms.py</code>’s <code>pure_1()</code> function passes in a list that grows in size (yellow line), while this algorithm just uses ‘<code>pass</code>’ without passing in anything.
</td></tr>
</table>

Room for error

Different calculations calculated at different speeds and processes running in the background may introduce variations to data.

Conclusions

- The best sorting algorithm from the ones discussed seems to be quick sort.
- Real life findings are not as stable as theory.
- Loading a composite datatype (such as list) into a function takes time, even if it’s not used.