CISC 233 Essential Algorithms Lab #3

Lab Description

Modify an existing codebase to run a series of tests to illustrate the big-O efficiency of `bubble_sort`, `insertion_sort`, `selection_sort`, `merge_sort`, and `quick_sort`.

The code you are given has the basic structure for running a series of tests for the `bubble_sort` algorithm. Carefully examine the code to understand what each part does.

Notes

- The program outputs to a CSV file – don't change the name or location of that file, EXCEPT you should change the `STUDENT_LASTNAME` global near the top of the source file to your own last name, which will then be incorporated into the output filename.
- There is a `VERBOSE_OUTPUT` mode that produces some output to the console – you can turn that on or off by changing the value of the global at the top of the file
- The program uses `random.sample` from the `random` module to generate the random arrays to be sorted.
- NOTE: for this lab, *your code must be written for Python 3.8 or above*. There were some significant changes between Python 2.x and Python 3.x, and code written for one version may not work for another. See Python Version Help in Canvas for more information.
- Your code should be fully commented, including:
    - updated file header comment with your name & date for modifications
    - references to original sources to any code you get from online (i.e. sort methods)
    - comments explaining any code that is not easily understandable

Introduction

- Download these instructions along with the python source file named `sorttest.py`.
- Carefully examine the code in `sorttest.py` to understand what each part does.
- Find the following lines of code near the top of `sorttest.py` and replace `yourlastname` with your actual last name:

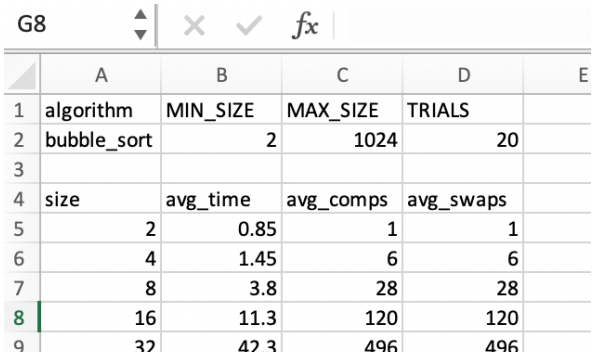
```
# Modifications by yourlastname

...
STUDENT_LASTNAME = 'yourlastname'
```

- Run the code, then find and examine the output file. It should be located in the same directory as the source file, with a name something like `rohrbaugh_2021-02-17_10-45-56.csv`.

Task 1

- Modify the code so that, instead of just counting and recording steps, we separately count and record *comparisons* and *swaps.*
- To do this, introduce two new global variables `comp_count` and `swap_count`, and modify the `bubble_sort` algorithm so that it increments `comp_count` whenever two elements of the list are compared, and increments `swap_count` whenever two elements are swapped.
- Also change the csv file output so that the main data section includes four columns instead of three (rather than `avg_steps`, it should have `avg_comps` and `avg_swaps`).
- For `bubble_sort`, since each comparison can result in a swap, we expect the two to grow at a similar rate.

Format of the output file from Task 1 should look like this (actual number values will vary):

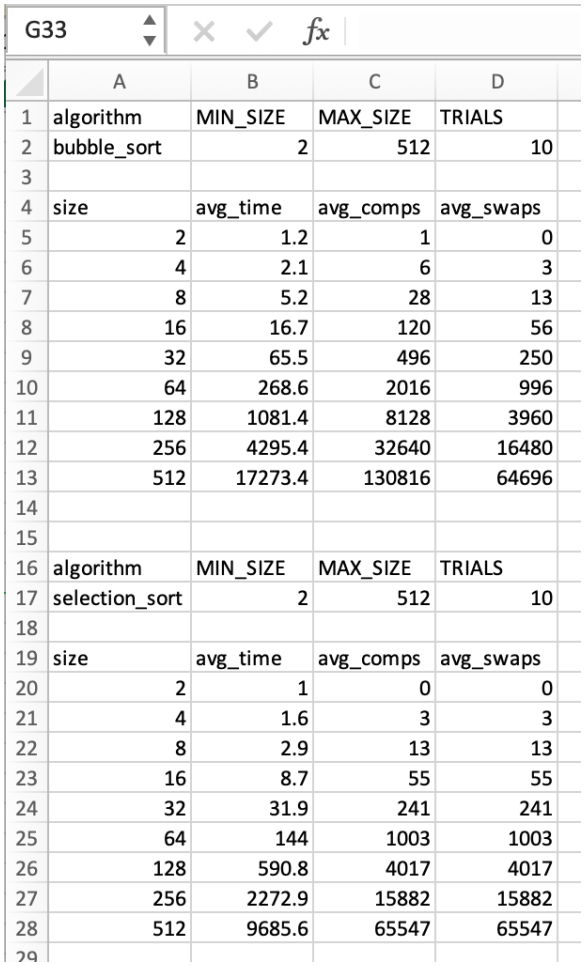


G8

	A	B	C	D	E
1	algorithm	MIN_SIZE	MAX_SIZE	TRIALS	
2	bubble_sort	2	1024	20	
3					
4	size	avg_time	avg_comps	avg_swaps	
5	2	0.85	1	1	
6	4	1.45	6	6	
7	8	3.8	28	28	
8	16	11.3	120	120	
9	32	42.3	496	496	

Task 2

- Add `selection_sort(list)` and `insertion_sort(list)` algorithms (write your own, or modify code you find online – in the latter case make sure to credit your source in the Python source file and in your lab report).
- Similar to the existing `bubble_sort`, `selection_sort` and `insertion_sort` should increment `comp_count` each time two elements from the list are compared, and increment `swap_count` each time two elements are swapped.
- Modify the `main()` method so that it tests all three sort methods (`bubble_sort`, `selection_sort`, and `insertion_sort`) with all output going to the same single output file. Insert two blank rows between the results for different sort algorithms, so that the output file will look like this (the structure, that is; though the actual numbers will differ):



G33

	A	B	C	D
1	algorithm	MIN_SIZE	MAX_SIZE	TRIALS
2	bubble_sort	2	512	10
3				
4	size	avg_time	avg_comps	avg_swaps
5	2	1.2	1	0
6	4	2.1	6	3
7	8	5.2	28	13
8	16	16.7	120	56
9	32	65.5	496	250
10	64	268.6	2016	996
11	128	1081.4	8128	3960
12	256	4295.4	32640	16480
13	512	17273.4	130816	64696
14				
15				
16	algorithm	MIN_SIZE	MAX_SIZE	TRIALS
17	selection_sort	2	512	10
18				
19	size	avg_time	avg_comps	avg_swaps
20	2	1	0	0
21	4	1.6	3	3
22	8	2.9	13	13
23	16	8.7	55	55
24	32	31.9	241	241
25	64	144	1003	1003
26	128	590.8	4017	4017
27	256	2272.9	15882	15882
28	512	9685.6	65547	65547
29				

Task 3

So far, we've been using randomly populated arrays – now we'd like to expand our tests to include two other initial configurations: almost sorted and reverse sorted.

- Modify the code so that it tests each of the algorithms with three different initial data configurations. Keep the tests with randomly-populated array, as in the initial code base.
- Add code to create and test a nearly sorted array, which should work as follows: start with an already sorted array (you can use the output from the test on random array); then modify by randomly swapping 5% of the elements. Calculate 5% of the array size, then use that number to repeat the process of randomly selecting two indexes in the array and swapping their contents, leaving an array with approximately 95% of elements in order. After that – run the sort test and print the results to the output file.
- Add another field to the header lines printed, so that it now it looks like this:

algorithm	initial_configuration	MIN_SIZE	MAX_SIZE	TRIALS
bubble_sort	almost sorted	2	512	10

- Do the same for a reverse-sorted initial configuration. You can use the built-in Python sort method with the option "reverse=True" as follows: array.sort(reverse=True). Once you have the reverse-sorted array, then use the current sort algorithm under consideration to re-sort the data.
- By the end of Task 3, there should be 9 test output sections in the output file: three each for bubble_sort, selection_sort, and insertion_sort (random, almost-sorted, and reverse-sorted).

Task 4

Add and test algorithms for `merge_sort(list)` and `quick_sort(list)`. You can write these yourself or find algorithms online and modify them to include the comparison count and swap count code. Make sure the names match what is specified here exactly.

- By the end of task 4, your output file should contain 15 sections: five sort methods (`bubble_sort, insertion_sort, selection_sort, merge_sort,` and `quick_sort`; each with three testing configurations: random, nearly sorted, and reverse sorted.

Task 5 (OPTIONAL/BONUS)

Add and test algorithms for `shell_sort(list)`. Again, you can write the code yourself or find code online and modify it to include the comparison count and swap count code.

- For shell sort, use the sequence 1, 4, 13, 40, 121..., defined by the formula:

$$\frac{3^k - 1}{2}$$

Lab Report

Write up a lab report with the following sections:

- Title page with assignment title, your name, class (CISC 233) and date
- A **Description** section providing a brief overview of the lab assignment
- A **Limitations** section describing any part of your code that isn't working exactly as intended
- A **Results** section that includes the output from your tests for each of the sort methods and each of the initial data configurations
- A **Discussion** section that compares the various algorithms, describing the big-O efficiency based on the test results