CCDSALG Term 3, AY 2024 – 2025

Project 1 Documentation – Convex Hull (Application of Stack Data Structure & Sorting Algorithms)

DECLARATION OF INTELLECTUAL HONESTY / ORIGINAL WORK

We declare that the project that we are submitting is the product of our own work. No part of our work was copied from any source, and that no part was shared with another person outside of our group. We also declare that each member cooperated and contributed to the project as indicated in the table below.

Section	Names and Signatures	Task 1	Task 2	Task 3	Task 4	Task 5	Task 6
S13	Gonzales, Ulrich Maeko			X	X	X	X
S12	Ortha, Gian Lorenzo	X	X	X	X	X	
S13	Obar, David	X	X	X	X		

Fill-up the table above. For the tasks, put an 'X' or check mark if you have performed the specified task (see MCO1 specs for the detailed task descriptions). Don't forget to affix your e-signature after your first name.

1. FILE SUBMISSION CHECKLIST: put a check mark as specified in the 3rd column of the table below. Please make sure that you use the same file names and that you encoded the appropriate file contents. For the .h and .c source files: make sure to include the names of the persons who created the codes.

FILE	DESCRIPTION	Put a check mark ✔ below to indicate that you submitted a required file
`stack.h` `Gonzales, Obar`	stack data structure header file	✔
`stack.c` `Gonzales, Obar`	stack data structure C source file	✔
`sort.h` `Gonzales, Obar`	"slow" and "fast" sorting algo header file	✔
`sort.c` `Gonzales, Obar, Ortha`	"slow" and "fast" sorting algo C source file	✔
`geom.h` `Gonzales, Obar`	header file for geometry related functions	✔
`geom.c` `Gonzales, Obar, Ortha`	C source file for geometry related functions	✔
`timer.h` `Ortha`	header file for the timer structure and related functions	✔
`timer.c` `Ortha`	source file for the timer related functions	✔
`gs_helpers.c` `Gonzales, Obar, Ortha`	Contains all the helper functions for the two source code files of the Graham's Scan algorithm	✔
`graham_scan1.c` `Gonzales, Obar`	Graham's Scan algorithm slow version (using the "slow" sorting algorithm)	✔
`graham_scan2.c` `Ortha`	Graham's Scan algorithm fast version (using the "fast" sorting algorithm)	✔

main1.c **Gonzales, Obar**	main module for the "slow" version	✔
main2.c **Ortha**	main module for the "fast" version	✔
n64.txt to **n32768.txt**	10 sample input files (with increasing values of *n*)	✔
n64-[1-5]-[fast/slow].txt to **nn32768-[1-5]-[fast/slow].txt**	100 output files, each corresponding to one input file, 1 of 5 trials with the input file, and the trials using the fast or slow version of the algorithm	✔
19.PDF	The PDF file of this document	✔

2. Indicate how to compile your source files, and how to RUN your exe files from the COMMAND LINE. Examples are shown below highlighted in yellow. Replace them accordingly. Make sure that all your group members test what you typed below because I will follow them verbatim (copy/paste as is). I will initially test your solution using a sample input text file that you submitted. Thereafter, I will run it again using my own test data:

- How to compile from the command line
  C:\MCO> gcc -Wall main1.c -o main1.exe
  C:\MCO> gcc -Wall main2.c -o main2.exe

- How to run from command line
  C:\MCO>main1
  C:\MCO>main2

Next, answer the following questions:

a. Is there a compilation (syntax error) in your codes? (YES or NO). NO
   WARNING: the project will automatically be graded with a score of **0** if there is syntax error in any of the submitted source code files. Please make sure that your submission does not have a syntax error.

b. Is there any compilation warning in your codes? (YES or NO) NO
   WARNING: there will be a 1 point deduction for every unique compiler warning. Please make sure that your submission does not have a compiler warning.

3. How did you implement your stack data structures? Did you use an array or linked list? Why? Explain briefly (at most 5 sentences).

Being straight here our group decided to just use arrays for a number of reasons, but for the sake of concision with clear regards to the instructions above, and here are them. One, our group's more familiar with the inner workings of arrays, naturally this would be more ergonomic for us in making our stack data structures. Two, compared to linked lists, arrays tend to be less syntactically demanding when it comes to programming. Lastly and with relation to the 2nd aforementioned reason, linked lists demand pointers, something of which will be of added constraints when programming this project.

4. Disclose **IN DETAIL** what is/are NOT working correctly in your solution. Please be honest about this. NON-DISCLOSURE will result in severe point deduction. Explain briefly the reason why your group was not able to make it work.

For example:

The following are NOT working (buggy):

a.

b.

We were not able to make them work because:

a.

b.

5. Based on the exhaustive testing that you did, fill-up the Comparison Table below that shows the performance between the "slow" version versus the "fast" version. Test for 10 different values of *n*, starting with *n = 2^6 = 64* points.

Table 5a: Executions Times of Each Trial Trial in the "Slow" Version

	Execution Time (ms)									
Problem Size	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**	**16384**	**32768**
1	0.0230	0.0620	0.2330	0.8760	4.2000	8.1760	33.4870	114.6630	465.3720	2060.9960
2	0.0240	0.0660	0.2340	0.8420	5.9660	7.9120	30.6800	123.8140	477.2170	2461.3480
3	0.0230	0.0640	0.2250	0.8460	1.9330	7.3410	31.8960	135.9030	478.5690	2354.0510
4	0.0210	0.0600	0.2470	0.8450	2.9610	7.4890	34.8560	156.1260	472.2500	2377.6990
5	0.0200	0.0690	0.2240	0.9060	3.2740	7.1130	36.7340	120.7350	508.9200	2037.2510
Average	**0.0222**	**0.0642**	**0.2326**	**0.8630**	**3.6668**	**7.6062**	**33.5306**	**130.2482**	**480.4656**	**2258.2690**

Table 5b: Executions Times of Each Trial Trial in the "Fast" Version

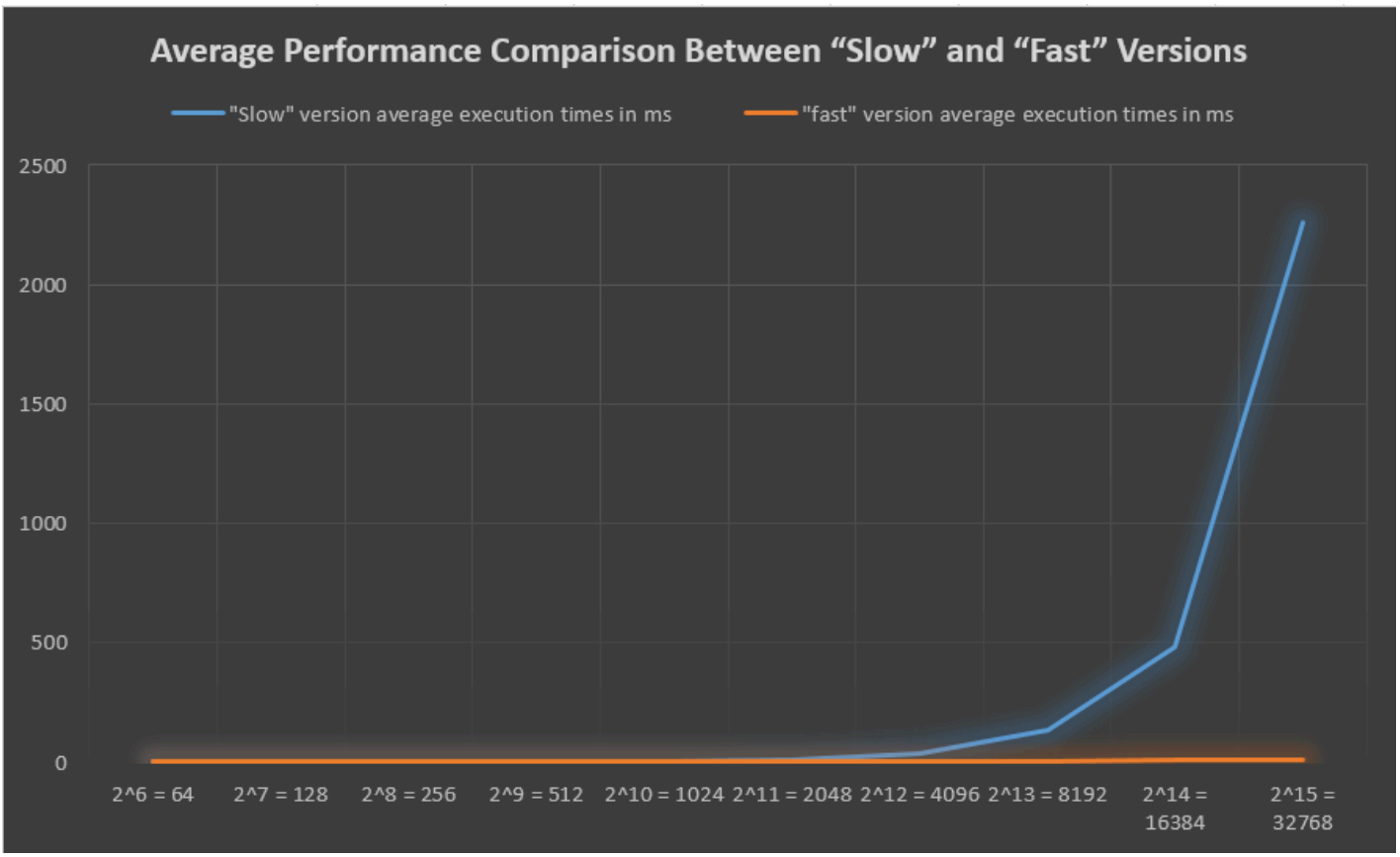
	Execution Time (ms)									
Problem Size	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**	**16384**	**32768**
1	0.0210	0.0350	0.0850	0.1750	0.3070	0.7900	1.1020	2.3400	4.6890	9.3270
2	0.0200	0.0380	0.0810	0.1670	0.2280	0.5770	1.3970	4.0660	6.5730	13.6840
3	0.0180	0.0380	0.0780	0.1730	0.3580	0.8330	1.7640	2.6100	6.7220	10.3130
4	0.0170	0.0350	0.0780	0.1740	0.3870	0.8140	1.9430	3.0180	6.8280	10.1480
5	0.0180	0.0390	0.0750	0.1950	0.3570	0.8270	1.9790	2.2280	4.8000	10.4420
Average	**0.0188**	**0.0370**	**0.0794**	**0.1768**	**0.3274**	**0.7682**	**1.6370**	**2.8524**	**5.9224**	**10.7828**

Table 5c: Average Performance Comparison Between "Slow" and "Fast" Versions

Test Case #	n (input size)	"Slow" version average execution time in *ms*	"Fast" version average execution time in *ms*
1	*2^6 = 64*	**0.022200 ms**	**0.018800 ms**
2	*2^7 = 128*	**0.064200 ms**	**0.036600 ms**
3	*2^8 = 256*	**0.233600 ms**	**0.079800 ms**
4	*2^9 = 512*	**0.863000 ms**	**0.176800 ms**
5	*2^10 = 1024*	**3.666800 ms**	**0.327400 ms**
6	*2^11 = 2048*	**7.606200 ms**	**0.768200 ms**
7	*2^12 = 4096*	**33.530600 ms**	**1.637000 ms**
8	*2^13 = 8192*	**130.248200 ms**	**2.852400 ms**
9	*2^14 = 16384*	**480.465600 ms**	**5.922400 ms**
10	*2^15 = 32768*	**2258.269000 ms**	**10.782800 ms**

NOTE: Make sure that you fill-up the table properly. It contributes 4 out of 15 points for the Documentation.

5. Create a graph (for example using Excel) based on the Comparison Table that you filled-up above. The x-axis should be the values of *n* and the y axis should be the execution time in milliseconds (ms). There should be two line graphs, one for the "slow" and the other for the "fast" data that should appear in one image. Copy/paste an image of the graph below.



NOTE: Make sure that you provide a graph based on your comparison table data above. It contributes 4 out of 15 points for the Documentation.

6. Analysis – compare and analyze the growth rate behaviors of the "slow" and "fast" versions based on the Comparison Table and the graphs above.

Answer the following question:

a. What do you think is the growth rate behavior of the "slow" version?
We think it has a growth rate behavior of Big Oh (n^2).

b. What do you think is the growth rate behavior of the "fast" version?
We think it has a growth rate behavior of Big Oh (n log n).

c. What do you think is/are the factor/s that make the "fast" version compute the results faster than the "slow" version?

For one thing, we thought that the fast version made a trade off between space and time, meaning that it took a lot more memory space than the slower version in order to deliver faster results. Another thing is that the fast version didn't necessarily have a 1 is to 1 approach when it comes to sorting everything, it had a more regrouping approach compared to the slow version, allowing it to save time and deliver the said results faster.

NOTE: Make sure that you provide cohesive answers to the three questions above. This part contributes 4 out of 15 points for the Documentation.

7. Fill-up the table below. Refer to the rubric in the project specs. It is suggested that you do an individual self-assessment first. Thereafter, compute the average evaluation for your group, and encode it below.

REQUIREMENT	AVE. OF SELF-ASSESSMENT
1. Stack	20 (max. 20 points)
2. Sorting algorithms	20 (max. 20 points)
3. Graham's Scan algorithm	40 (max. 40 points)
4. Documentation	15 (max. 15 points)
5. Compliance with Instructions	5 (max. 5 points)

TOTAL SCORE 100 over 100.

NOTE: The evaluation that the instructor will give is not necessarily going to be the same as what you indicated above. The self-assessment serves for your own reference only…