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
S12	Ramos, Marcus Timothy [illegible]	X	X	X	X	X	X
S12	Lusong, Keann Colin [illegible]		X				X
S12	Hung, Byron David [illegible]		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`	stack data structure header file	✓
`stack.c`	stack data structure C source file	✓
`sort.h`	"slow" and "fast" sorting algo header file	✓
`sort.c`	"slow" and "fast" sorting algo C source file	✓
`graham_scan1.c`	Graham's Scan algorithm slow version (using the "slow" sorting algorithm)	✓
`graham_scan2.c`	Graham's Scan algorithm fast version (using the "fast" sorting algorithm)	✓
`main1.c`	main module for the "slow" version	✓
`main2.c`	main module for the "fast" version	✓
`INPUT1.TXT` to `INPUT10.TXT`	10 sample input files (with increasing values of *n*)	✓
`OUTPUT1.TXT` to `OUTPUT10.TXT`	5 sample corresponding output files	✓
`GROUPNUMBER.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).

Our stack data structure uses an array to better accommodate the elements to be pushed in the stack. Since the elements that are pushed in the stack consist of two numbers, which are the x and y coordinates of a point in the graph, then we used an array that holds a structure value of two numbers to make it easier when dealing with them. An array is also easy to deal with in getting the index of the top element of the stack.

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.

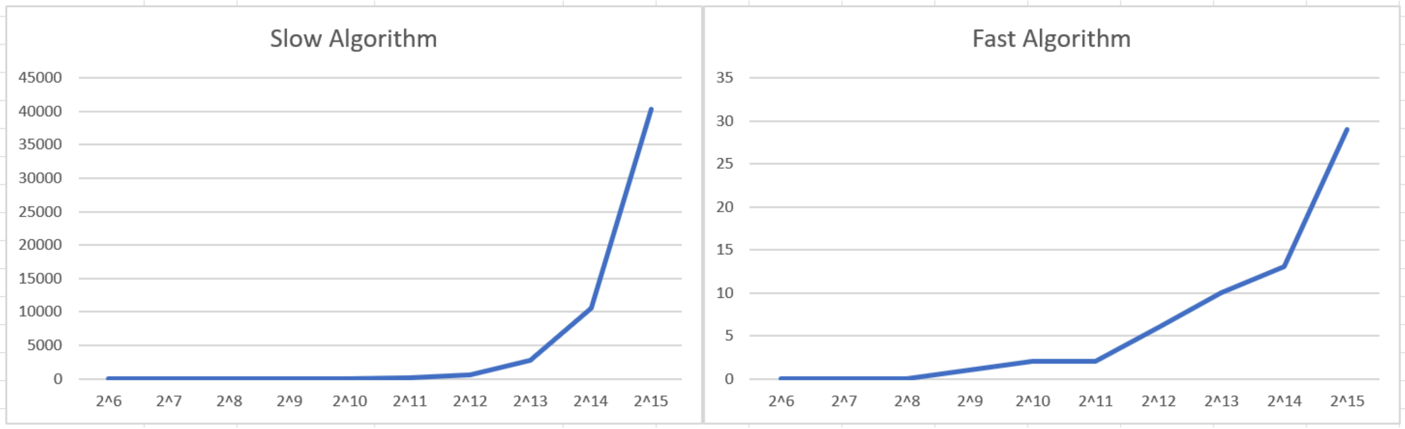
5. Based on the exhaustive testing that you did, fill-up the Comparison Table below that shows the performance between the "slow" version versus the "fast" version. Test for 10 different values of *n*, starting with *n = 2^6 = 64* points.

Table: Performance Comparison Between "Slow" and "Fast" Versions

Test Case #	n (input size)	"Slow" version: execution time in *ms*	"Fast" version: execution time in *ms*
1	*2^6 = 64*	1.000000	0.000000
2	*2^7*	1.000000	0.000000
3	*2^8*	4.000000	0.000000
4	*2^9*	12.000000	1.000000
5	*2^10*	41.000000	2.000000
6	*2^11*	188.000000	2.000000
7	*2^12*	626.000000	6.000000
8	*2^13*	2775.000000	10.000000
9	*2^14*	10519.000000	13.000000
10	*2^15 = 32768*	40296.000000	29.000000

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?

- Based on the comparison table and graph above, the growth rate behavior of the “slow” version is quadratic (n^2). As the input size doubles, the execution time increases approximately by a factor of 4. Furthermore, the graph shows a steep upward curve, normally shown in a quadratic growth.
- From 28 to 29 (256 to 512 points), time goes from 4ms to 12ms (approx. 3x).
- From 29 to 210 (512 to 1024 points), time goes from 12ms to 41ms (approx. 3.4x).
- From 210 to 211 (1024 to 2048 points), time goes from 41ms to 188ms (approx. 4.5x).
- From 211 to 212 (2048 to 4096 points), time goes from 188ms to 626ms (approx. 3.3x).
- From 212 to 213 (4096 to 8192 points), time goes from 626ms to 2775ms (approx. 4.4x).
- From 213 to 214 (8192 to 16384 points), time goes from 2775ms to 10519ms (approx. 3.7x).
- From 214 to 215 (16384 to 32768 points), time goes from 10519ms to 40296ms (approx. 3.8x).

b. What do you think is the growth rate behavior of the “fast” version?

- Based on the comparison table and graph above, the growth rate behavior of the “fast” version is linear-logarithmic $(n\,log\ n)$. As the input size doubles, the execution time increases approximately by a factor slightly larger than 2. Furthermore, the graph shows a relatively gentle upward curve, generally shown in a linear-logarithmic growth.
- From 29 to 210 (512 to 1024 points), time goes from 1ms to 2ms (2x).
- From 210 to 211 (1024 to 2048 points), time goes from 2ms to 2ms (1x, likely due to measurement granularity at very low times).
- From 211 to 212 (2048 to 4096 points), time goes from 2ms to 6ms (3x).
- From 212 to 213 (4096 to 8192 points), time goes from 6ms to 10ms (approx. 1.67x).
- From 213 to 214 (8192 to 16384 points), time goes from 10ms to 13ms (approx. 1.3x).
- From 214 to 215 (16384 to 32768 points), time goes from 13ms to 29ms (approx. 2.23x).

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

- The biggest factor that makes the “fast” version compute the results much faster than the “slow” version is the sorting algorithm used in each iteration. The “slow” version uses the bubble sort algorithm, which has a time complexity of O(n^2n), which is significantly slower than the faster merge sort algorithm, which has a time complexity of O(n log n). Sorting the data takes up a huge chunk of the process in the Graham Scan algorithm, which means it also dictates the computing time of the majority of the program.

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	_19_ (max. 20 points)
2. Sorting algorithms	_17_ (max. 20 points)
3. Graham's Scan algorithm	_38_ (max. 40 points)
4. Documentation	_14_ (max. 15 points)
5. Compliance with Instructions	_4_ (max. 5 points)

TOTAL SCORE _92_ 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…

サルバドール・フロランテ