**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** |
| S11 | Agunanne, Henry | X | X | X | X | X | X |
| S11 | Inocencio, Chazwick | X | X | X | X | X | X |
| S11 | Lu, Qinpei | X | X | 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** | 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 graham\_scan1.c sort.c stack.c -o main1.exe**  C:\MCO> **gcc -Wall main2.c graham\_scan2.c sort.c stack.c -o main2.exe**   * How to run from command line   C:\MCO>main1  C:\MCO>main2    Next, answer the following questions:   1. 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.   1. 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).  **We implemented our stack using a fixed-size array. This decision was based on the fact that the maximum number of points was predefined (MAX\_POINT\_COUNT), so memory requirements were known in advance. Using an array allowed for faster access times and reduced overhead compared to a linked list. It also simplified our implementation since dynamic memory management wasn't required. Given the application's performance sensitivity (used in Graham Scan), this approach was efficient and reliable.** |
| 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. The first two lines of the output using the ***sample-input.txt*** file is different from the expected output file ***sample-output.txt*** for both the fast and slow algorithms  b.  We were not able to make them work because:  a. We could not figure out the exact part of our program’s logic that was causing the output disparity. We believe It might be the ***polarArr[]*** sorting logic but we weren’t able to come up with the implementation fix.  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* | 0.048 | 0.038 | | 2 | *2^7 = 128* | 0.131 | 0.065 | | 3 | *2^8 = 256* | 0.427 | 0.131 | | 4 | *2^9 = 512* | 1.644 | 0.228 | | 5 | *2^10 = 1024* | 6.112 | 0.473 | | 6 | *2^11 = 2048* | 17.271 | 1.163 | | 7 | *2^12 = 4096* | 51.259 | 2.271 | | 8 | *2^13 = 8192* | 160.411 | 6.371 | | 9 | *2^14 = 16384* | 623.547 | 11.579 | | 10 | *2^15 = 32768* | 2613.451 | 17.886 |   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?  **The “slow” version (using bubble sort) has a time complexity of O(n²) due to the sorting step. As the input size increases, the execution time grows quadratically, leading to “slow” performance on large datasets.**  b. What do you think is the growth rate behavior of the “fast” version?  **The “fast” version (using heap sort) has a time complexity of O(n log n)**. **This makes it significantly faster for large input sizes compared to the slow version, particularly when n reaches thousands of points.**  c. What do you think is/are the factor/s that make the “fast” version compute the results faster than the “slow” version?  **The primary factor is the use of an efficient sorting algorithm. Bubble sort is inefficient for large datasets due to repeated comparisons and swaps. Heap sort significantly reduces the number of operations needed to sort the points by their polar angle, making the overall convex hull algorithm more scalable.**  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 | 15     (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 | 95 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-ル・フロランテ***