ECE505 Project Part 1

Bharath Santhosh

Problem Description

Create a Convex Hull algorithm: an algorithm that can determine the smallest (in dimension) convex polygon that encloses all the points in a set of 2D in a Cartesian place. The program will take input of points from an external CSV format file that will have each point separated into different subsequent lines and it should output the points that make up the polygon.

Method 1: Jarvis' Algorithm

Create a class that reads the file as input and validates it

a) Create a function of type Boolean that determines if the file exists or not. It will take the proposed directory name to the input file
    i. Attempt to open the file using the filename provided
    ii. If it can't be found, return False and set a File Not Found Error Flag.
    iii. If it is found, return True
b) Create a function of type Boolean that checks if the file can be read. It will take the proposed directory name to the input file
    i. Open and attempt to read the first line of the file
    ii. If an error occurs during reading, return False and set an Input/Output Error Flag
    iii. If the file can be read successfully, return True
c) Create a function of type Boolean that checks if the file is empty. It will take the proposed directory name to the input file
    i. Open the file and read all content
    ii. Remove all whitespace and newline characters from the content
    iii. If the remaining content has a length of zero, return False and set an"Empty File Error" Flag
    iv. Otherwise, return True
d) Create a function that returns a Boolean and validates coordinate format. It will take an input from a single line from the file
    i. Check if the line contains a comma (for identifying CSV)
    ii. Split the line by comma and count the number of values
    iii. Loop through each value in the string
        a. Remove all whitespaces
        b. Check if the value is numeric

c. If not, return False and set an “Invalid Data Format Error” Flag

iv. If the loop is cleared, return True

e) Create a function of type Boolean that checks if the dimensions are correct. This takes the contents of the whole file.

i. Loop through all the lines in the array

a. Read the number of comma-separated values in the line

b. If the count is not 2, return False and set a “Dimension Mismatch Error” Flag (3 dimensions can be done with other algorithms)

ii. If cleared, return True

f) Create a function of type Integer and sets up the file into point coordinates. The function takes in one string parameter (filename) and one array parameter (to store points)

i. Open the file

ii. Create a counter variable initialized to zero for tracking number of points

iii. Loop through each line in the file

a. Split the line by comma

b. Create a tuple with each value in sequence. This is the point

c. Add the point to the array

d. Increment the counter

iv. Return the counter representing total number of points parsed

Create a class to handle duplicate point filtering and validation

a) Create a function of type Boolean and removes duplicate points. The function takes in one array parameter (all points) and one integer parameter (number of points)

i. Create a new temporary array to store unique points

ii. Create a counter for unique points initialized to zero

iii. Loop through each point in the original array using index i from 0 to number of points minus 1

a. Create a Boolean flag "isDuplicate" initialized to False

b. Loop through the temporary array using index j from 0 to counter minus 1

A. If the x and y coordinates at point i is equal to those in point j, set isDuplicate to true and break

c. If isDuplicate is false, add the point at index i to the temporary array and increment counter (the repeated ones will be saved deeper down in the list, while only saving one of them)

iv. Copy all points from the temporary array back to the original array

v. Return True if the counter of unique points is greater than or equal to 3, otherwise return False and set an "Insufficient Points Error" Flag.

b) Create a function of type Boolean that checks if all points are collinear. The function takes in one array parameter (points array) and one integer parameter (number of points)

i. If number of points is less than 3, return true (collinear by default)

ii. Save a Boolean value “isCollinear” as True

iii. Find the absolute slope between the first two points and save it

iv. Loop through all the points afterwards

a. Find the absolute slope of each point with the first point.

b. If the slope is different than the one found first (use a difference of a small value), set isCollinear to False and break

v. If isCollinear is True, return True and set a “Collinear Point Error” Flag

vi. Otherwise, return False

Create a class to implement Jarvis’ algorithm

a) Create a struct that carries two floating point values, x and y, for each coordinate in a 2D point

b) Create a function of Integer type and finds the leftmost point. The function takes in one array parameter (points array) and one integer parameter (number of points)

i. Create a variable "leftmost" initialized to 0 (first point) to store the index

ii. Loop through points from index 1 (second point) to the final index

a. If the x-coordinate of the current point is less than the x-coordinate of the point at the leftmost index, update leftmost to the current index.

b. **Otherwise**, if the x-coordinate of the current point equals the x-coordinate of the point at the leftmost index and y-coordinate of the current point is less than y-coordinate of the point at the leftmost index, update leftmost to the current index

iii. Return the index

c) Create a function on Integer type returning orientation of three points. The function takes in three point parameters: p, q, and r

i. Calculate the value: (q.y minus p.y) multiplied by (r.x minus q.x) minus (q.x minus p.x) multiplied by (r.y minus q.y) (*Dot here is the coordinates marker; q.y is y-coordinate of q, taken from the struct)

ii. If the value is equal to zero, return 0 (collinear points, only among these three points)

iii. If the value is greater than zero, return 1 (clockwise orientation)

iv. If the value is less than zero, return 2 (counterclockwise orientation)

d) Create an Integer type function that computes the convex hull using the Jarvis Algorithm. The function takes in three parameters: points array, number of points, and hull array to store the result
    i. If the number of points is less than 3, return 0 (cannot form a hull)
    ii. Call the function to find the leftmost point and store the result in a variable
    iii. Save the vertex in the leftmost index as the current vertex
    iv. Create a counter variable to find the size of the hull
    v. Create a loop that continues until the current vertex becomes equal to the leftmost vertex (after first iteration, do-while format)
        a. Add the point at the current vertex index to the hull array at the index given by the hull counter variable
        b. Increment the hull counter variable
        c. Create a variable for the next point and initialize it to the current point index plus 1 modulo number of points
        d. Loop through all the points using indices from 0 to the number of points minus 1
            A. Call the orientation function passing points at the current point, index point, and next point
            B. If the orientation equals 2 (counterclockwise), update the next point to the index point
            C. Else if the orientation equals 0 (collinear), check if the euclidean distance from the current point to the index point is greater than the euclidean distance from the current point to the next point. If so, update the next point to the index point.
        e. Set the current point to be equal to the next point
    vi. While the current point does not equal the leftmost point
    vii. Return the hull counter
e) Create a Float type function that calculates the euclidean distance between two points. The function takes in two parameters: the first and second points
    i. Subtract the x and y coordinates of the first point with the x and y coordinates of the second point, each.
    ii. Return the whole square root of x-coordinate difference squared plus the y-coordinate difference squared
f) Create a function that prints the convex hull results. The function takes in two parameters: hull array and hull size
    i. Print header message: "Convex Hull Vertices (in order):"
    ii. Loop through the hull array from index 0 to the end
        a. Print the point at current index in format: "Point index: (x-coordinate, y-coordinate)"

iii. Print footer message: "Dimension of the convex hull: {hullSize}"

g) Create a function that writes the convex hull results to output file. The function takes in three parameters: hull array, hull size, and output file name

   i. Open the output file for writing

   ii. If the file cannot be opened, display error message and return

   iii. Write header to file: "Convex Hull Vertices (in order):"

   iv. Loop through the hull array from index 0 to the end

      a. Write the point at current index to file in format: "Point index: (x-coordinate, y-coordinate)"

   v. Write footer to file: "Dimension of the convex hull: {hullSize}"

   vi. Close the output file

   vii. Display success message: "Results written to {file name}"

Main Function

a) Declare variables for input filename and output filename

   i. Prompt the user to enter the input and output CSV filenames

   ii. Store the input in their respective variables

b) Create an object of the file validation class

c) Call the file existence checking function

   i. If the function returns false, display "Error: File Not Found" and exit program

d) Call the file readability checking function

   i. If the function returns false, display "Error: Input/Output Error - Cannot read file" and exit program

e) Call the empty file checking function

   i. If the function returns false, display "Error: Empty File" and exit program

f) Read all lines from the file into a temporary array

g) Call the dimension consistency checking function passing the array of lines

   i. If the function returns false, display "Error: Dimension Mismatch – Either Inconsistent Dimensions or non-2D Dimensions" and exit program

h) Validate each line format by calling the coordinate format validation function

   i. Loop through each line in the temporary array

      a. Call the validation function for current line

    b. If the function returns false, display "Error: Invalid Data Format - Non-numeric or unexpected coordinates" and exit program

i) If all validations pass, create an array to store points

j) Call the function to parse file and populate points array, store returned count

k) Create an object of the duplicate filtering class

l) Call the duplicate removal function

    i. If the function returns false, display "Error: Insufficient Points - At least 3 unique points required" and exit program

m) Call the collinearity checking function

    i. If the function returns true, display "Error: Collinear Points - All points lie on a single line" and exit program

n) Create an object of the Jarvis' algorithm class

o) Create an empty array to store the convex hull result

p) Call the convex hull computation function

    i. Pass in the points array, number of points, and hull result array

    ii. Store the returned hull size in a variable

q) Call the function to print results to screen

    i. Pass in the hull array and hull size

r) Call the function to write results to output file

    i. Pass in the hull array, hull size, and output filename

s) Display program completion message: "Convex hull computation completed successfully"

t) Exit program

Method 2: Graham Scan

Create a class that reads the file as input and validates it

a) Create a function of type Boolean that determines if the file exists or not. It will take the proposed directory name to the input file

    i. Attempt to open the file using the filename provided

    ii. If it can't be found, return False and set a File Not Found Error Flag.

    iii. If it is found, return True

b) Create a function of type Boolean that checks if the file can be read. It will take the proposed directory name to the input file

    i. Open and attempt to read the first line of the file

    ii. If an error occurs during reading, return False and set an Input/Output Error Flag

iii. If the file can be read successfully, return True

c) Create a function of type Boolean that checks if the file is empty. It will take the proposed directory name to the input file

i. Open the file and read all content

ii. Remove all whitespace and newline characters from the content

iii. If the remaining content has a length of zero, return False and set an"Empty File Error" Flag

iv. Otherwise, return True

d) Create a function that returns a Boolean and validates coordinate format. It will take an input from a single line from the file

i. Check if the line contains a comma (for identifying CSV)

ii. Split the line by comma and count the number of values

iii. Loop through each value in the string

a. Remove all whitespaces

b. Check if the value is numeric

c. If not, return False and set an “Invalid Data Format Error” Flag

iv. If the loop is cleared, return True

e) Create a function of type Boolean that checks if the dimensions are correct. This takes the contents of the whole file.

i. Loop through all the lines in the array

a. Read the number of comma-separated values in the line

b. If the count is not 2, return False and set a “Dimension Mismatch Error” Flag (3 dimensions can be done with other algorithms)

ii. If cleared, return True

f) Create a function of type Integer and sets up the file into point coordinates. The function takes in one string parameter (filename) and one array parameter (to store points)

i. Open the file

ii. Create a counter variable initialized to zero for tracking number of points

iii. Loop through each line in the file

a. Split the line by comma

b. Create a tuple with each value in sequence. This is the point

c. Add the point to the array

d. Increment the counter

iv. Return the counter representing total number of points parsed

Create a class to handle duplicate point filtering and validation

a) Create a function of type Boolean and removes duplicate points. The function takes in one array parameter (all points) and one integer parameter (number of points)
    i. Create a new temporary array to store unique points
    ii. Create a counter for unique points initialized to zero
    iii. Loop through each point in the original array using index i from 0 to number of points minus 1
        a. Create a Boolean flag "isDuplicate" initialized to False
        b. Loop through the temporary array using index j from 0 to counter minus 1
            A. If the x and y coordinates at point i is equal to those in point j, set isDuplicate to true and break
        c. If isDuplicate is false, add the point at index i to the temporary array and increment counter (the repeated ones will be saved deeper down in the list, while only saving one of them)
    iv. Copy all points from the temporary array back to the original array
    v. Return True if the counter of unique points is greater than or equal to 3, otherwise return False and set an "Insufficient Points Error" Flag.
b) Create a function of type Boolean that checks if all points are collinear. The function takes in one array parameter (points array) and one integer parameter (number of points)
    i. If number of points is less than 3, return true (collinear by default)
    ii. Save a Boolean value “isCollinear” as True
    iii. Find the absolute slope between the first two points and save it
    iv. Loop through all the points afterwards
        a. Find the absolute slope of each point with the first point.
        b. If the slope is different than the one found first (use a difference of a small value), set isCollinear to False and break
    v. If isCollinear is True, return True and set a “Collinear Point Error” Flag
    vi. Otherwise, return False

Create a class to implement a stack data structure for points

a) Create an array to store points with fixed maximum size
b) Create an integer initialized to negative 1 to track the top of the stack
c) Create a function that pushes a point to the stack

i. increment the stack top by 1

ii. Store the point parameter at the top spot in the array

d) Create a function that pops a point and returns

i. If top is less than 0, return an error

ii. read and store the value at the top index

iii. reduce the top integer by 1

iv. return the saved value

e) Create a function to peek the top point

i. Perform everything in d) outside of reducing the top integer (iii.)

f) Create a function that returns the second-to-the-top point without removing it

i. If the top is less than 1, return an error

ii. Perform e) but by finding top – 1

g) Create a function that returns the current size of the stack

i. Return top + 1

Create a class to implement the Graham Scan algorithm

a) Create a struct that carries two floating point values, x and y, for each coordinate in a 2D point

b) Create an Integer type function that finds the bottommost point, leftmost in case of a tie. The function takes one array parameter (all points) and one integer parameter (number of points)

i. Create a variable meant to store the lowest point's index, and it should get the index of the first value in the point list.

ii. Loop through each point in the array parameter after the first one by index

a. If the y-coordinate of the current point in the loop is less than the lowest point's y-coordinate, overwrite the lowest point with the current one.

b. Otherwise, if the y-coordinates are equal and the x-coordinate of the current point is less than the x-coordinate of the lowest point, overwrite the lowest point with the current one.

iii. Return the lowest point's index

c) Create a Float type function that calculates the polar angle. It takes two parameters: the anchor point and the target point

i. Subtract the x and y coordinates of the target with the x and y coordinates of the anchor, each.

ii. Return the radian arctangent of the y-coordinate difference divided by the x-coordinate difference

d) Create a Float type function that calculates the euclidean distance between two points. The function takes in two points
    i. Subtract the x and y coordinates of the first point with the x and y coordinates of the second point, each.
    ii. Return the whole square root of the x-coordinate difference squared plus the y-coordinate difference squared
e) Create a void function that swaps two points in an array. It takes three parameters: the full points array and the two integer indices to be swapped
    i. Store the point in the first index in a temporary variable
    ii. Set the first index in the points array to be equal to the point in the second index
    iii. Set the second index in the points array to be equal to the temporary variable
f) Create a void function that sorts the points by polar angle with respect to the anchor point. It takes three parameters: the full points array, the number of points, and the integer anchor point index
    i. Call the swap function to move the anchor point to index 0 in the array
    ii. Loop through the array from after the first index (anchor point) with index i
        a. Loop a second time starting from after the index at this loop (i+1) with index j
            A. Find and store the polar angle of the points at index i and the anchor point
            B. Repeat for index j and the anchor
            C. If the first is greater than the second, call the swap function to swap the points at i and j.
            D. If they are equal (or in the range of a very small value)
                1. Get the euclidean distance from anchor to i and anchor to j
                2. If the first is greater than the second, swap them
    iii. Function end
g) Create an Integer function that finds the orientation of three points. It takes the three points as parameters: p, q, and r.
    i. Find this result: (q.y minus p.y) multiplied by (r.x minus q.x) minus (q.x minus p.x) multiplied by (r.y minus q.y)
    ii. If the absolute value of the result is less than a very small value or near to zero, return 0 as the points are collinear.
    iii. If the value is greater than zero, return 1 (clockwise)
    iv. If the value is less than zero, return 2 (counterclockwise)
h) Create a function that computes the convex hull using Graham Scan. The function takes in three parameters: points array, number of points, and hull array to store result

i. If number of points is less than 3, return 0 (cannot form a hull)

ii. Call the function to find the bottommost point and store the result in variable "anchorIndex"

iii. Call the sorting function to sort all points by polar angle relative to the anchor point

iv. Create a stack object using the stack class

v. Push the first point (anchor point) onto the stack

vi. Push the second point onto the stack

vii. Push the third point onto the stack

viii. Loop through remaining points from index 3 to number of points minus 1

a. Create a while loop that continues while the orientation of next-to-top, top, and current point is not counterclockwise

A. Call the orientation function with next-to-top point, top point, and current point

B. If orientation does not equal 2 (not counterclockwise), pop the stack

b. After the while loop, push the current point onto the stack

ix. Create a variable "hullSize" and set it equal to the size of the stack

x. Loop through the stack from bottom to top

a. Pop each point from the stack and store in hull array in reverse order (so hull array has points in correct order)

xi. Return hullSize

i) Create a function that prints the convex hull results. The parameters are: final hull array and hull size

i. Print header message "Convex Hull Vertices (in order):"

ii. Loop through the hull array from index 0 to the end

a. Print the point at current index in format: "Point index: (x-coordinate, y-coordinate)"

iii. Print footer message: "Dimension of the convex hull: {hullSize}"

j) Create a function that writes the convex hull results to output. There are three parameters: hull array, hull size, and output file name

i. Open the output file for writing

ii. If the file cannot be opened, display error message and return

iii. Write header to file: "Convex Hull Vertices (in order):"

iv. Loop through the hull array from index 0 to the end

a. Write the point at current index to file in format: "Point index: (x-coordinate, y-coordinate)"

v. Write footer to file: "Dimension of the convex hull: {hullSize}"

vi. Close the output file

vii. Display success message: "Results written to {file name}"

k) Create a function to handle collinear points at the end of the sorted array. The function pas two parameters: the full points array and the number of points.

   i. Create a variable "i" initialized to number of points minus 2

   ii. Create a while loop that continues while i is greater than or equal to 0

      a. Call orientation function with anchor point (at index 0), point at index i, and last point (at index number of points minus 1)

      b. If orientation equals 0 (collinear), decrement I

      c. If orientation does not equal 0, break from loop

   iii. Increment i by 1 (i now points to first collinear point with the last point)

   iv. If i is less than number of points minus 1 (meaning there are collinear points at the end)

      a. Reverse the order of points from index i to number of points minus 1

         A. Create left pointer initialized to I

         B. Create right pointer initialized to number of points minus 1

         C. While left is less than right

            1. Swap points at left and right indices

            2. Increment left

            3. Decrement right

Main Function

a) Declare variables for input filename and output filename

   i. Prompt the user to enter the input and output CSV filenames

   ii. Store the input in their respective variables

b) Create an object of the file validation class

c) Call the file existence checking function

   i. If the function returns false, display "Error: File Not Found" and exit program

d) Call the file readability checking function

   i. If the function returns false, display "Error: Input/Output Error - Cannot read file" and exit program

e) Call the empty file checking function

   i. If the function returns false, display "Error: Empty File" and exit program

f) Read all lines from the file into a temporary array

g) Call the dimension consistency checking function passing the array of lines
    i. If the function returns false, display "Error: Dimension Mismatch – Either Inconsistent Dimensions or non-2D Dimensions" and exit program
h) Validate each line format by calling the coordinate format validation function
    i. Loop through each line in the temporary array
        a. Call the validation function for current line
        b. If the function returns false, display "Error: Invalid Data Format - Non-numeric or unexpected coordinates" and exit program
i) If all validations pass, create an array to store points
j) Call the function to parse file and populate points array, store returned count
k) Create an object of the duplicate filtering class
l) Call the duplicate removal function
    i. If the function returns false, display "Error: Insufficient Points - At least 3 unique points required" and exit program
m) Call the collinearity checking function
    i. If the function returns true, display "Error: Collinear Points - All points lie on a single line" and exit program
n) Create an object of the Graham Scan algorithm class
o) Create an array to store the convex hull result
p) Call the Graham Scan convex hull computation function
    i. Pass in the points array, number of points, and hull result array
    ii. Store the returned hull size in a variable
q) Call the function to print results to screen
    i. Pass in the hull array and hull size
r) Call the function to write results to output file
    i. Pass in the hull array, hull size, and output filename
s) Display program completion message: "Convex hull computation completed successfully”
t) Exit program

Algorithm Analysis

For each of these algorithms, collinearity is a significant breaker. If all of the points in the plot are in a single line, there is no shape that can be drawn with them. As such, each instance of collinearity will be treated as erroneous.

Jarvis’ Algorithm works by first identifying the leftmost point in the given vertices, then ‘wrapping’ the line around the points by finding the next and farthest outermost point that won’t leave any

point outside. This is done by identifying the orientation of the angle connecting three points, of which one end will be the leftmost point in the first instance. If the orientation of the three points (direction of the second half of the line formed between the three, starting with the current point) is counter-clockwise, the middle point will be shifted to become the end point. If the three points are collinear, the furthest point will be chosen as the middle point.

Graham Scan is done from a similar comparison of three points to Jarvis' Algorithm, but the starting point will be identified from the bottommost, then leftmost index in the plot. It also finds the orientation between three points, but the initial sorting of the points allows it to immediately access adjacent points for the hull. It identifies concave spots in the traversal by observing the orientation trend. If a concave curve shows up, the center point from which the concave appeared will be removed from the hull so that the shape will lose one dimension. This is repeated until the smallest dimension convex hull is found.

Criteria	Jarvis' Algorithm	Graham Scan
Basic Operation	Perform orientation tests between three points to identify the overall shape of the hull	Perform orientation tests between three points to identify the overall shape of the hull
Initial Point	Leftmost point	Bottommost point
Hull Identification	Set the first two points of the orientation check within a loop. If any of the third points can lead to a counterclockwise orientation, substitute the second point with that third one.	Pre-sort the points according to the angle between them and the bottommost point. With this done, find the orientation of three points segments in the sorted list. If the angle is not counterclockwise, the middle point is discarded from the hull.
Number of comparisons	Comparisons done for each point in the vertex, repeated for each point in the hull.	Sorting will take a double nested loop, but the internal one starting from the outer one's current value (n log n). Hull operation is done by iterating through each point in the sorted list.
Average Time Complexity	O(n*h) where n is the number of points and h is the number of hull vertices	n logn + n -> O(n logn).
Worst case Time Complexity	$O(n^2)$ when hull vertices equals total vertices	O(n log n)
Space complexity	O(n) for each point and hull output. Two cases	O(n) in complexity, but repeated for the points, the stack and the hull output. Three cases
Dependency on hull size	Yes, the time complexity asks for it as well	No, the hull size will not change the time complexity here
Optimal Case	Only when the hull size is very	General case and when hull size

	small	is closer to the number of vertices

Algorithm Recommendation

Upon reading the previous table, the recommendation should be obvious. The preferred general choice is Graham Scan.

Jarvis' Algorithm can only be quicker than Graham Scan if the hull size is significantly smaller than the number of vertices ($h < \log n$). This is a rare case, so generally the Graham Scan method will be the preferred option, especially in terms of reliability. You will be able to accurately know the time the Graham Scan algorithm will take if you are aware of your computer's operation speed and the full detailed operations of the algorithm, only while knowing the number of points in the data. For Jarvis' Algorithm, however, since the hull size can only be found from the algorithm, the time taken cannot be predicted early.

Space may be a point in Jarvis' Algorithm's favor, but even that is very minimal. The space complexities of both algorithms are identical. It is only the constant multipliers in the space requirement that is slightly higher in the Graham Scan algorithm. As such, if the user is so starved of memory that the Jarvis' Algorithm is filling it up, Graham Scan will cause an overflow. However, this is a really low bar and should not be generally considered.

One can argue that Jarvis' Algorithm is the simpler of the two as well. Since the number of functions within the classes are smaller in this one compared to Graham Scan, it can be considered easier to implement. However, as a software engineer, this is not a point that should be considered as the client would much prefer the faster algorithm regardless of simplicity.

As such, Graham Scan is the best choice between the two in the general case.

Metrics

- Time Based Metrics
  - Preprocessing Time: All of the processes before the main algorithm begins will be times. As the error handling and initial steps are identical between the two algorithms, this should be identical for both as long as the input file is the same
  - Algorithm Time: Total time for the execution of the algorithm. This should show the Graham Scan algorithm being faster given that the provided data is sufficiently large.
  - Total Runtime: Sum of both the above and some other internal processes. This can be demonstrated to show the efficiency of the full algorithm
  - Measurement Means: Wall Clock algorithms within the used programming language, like chrono in C++ and time() in Python. Will have measurements set for each test shown above
- Operation Count Metrics
  - Orientation tests: Add an additional counter to the orientation test function that returns the number of times it was called.

    - Number of comparisons: In each algorithm, count the number of times the most internal loop has been run.
    - Measurement Means: Either a global variable or an extra parameter in each function, most likely the former. The variable will be incremented by 1 in the mentioned equation
- Space Metrics
    - Peak Memory Usage: By the end of the algorithm, count how many memory spaces were used
    - Stack measurement (Graham Scan only): Keep track of the maximum size of the stack
    - Measurement Means: Add up all of these sizes taken from the created variables and arrays.
- Input Output Characterization Metrics
    - Input Size after Removal of Duplicates: Find the difference between the input and output of the duplicate removal algorithm.
    - Hull Size: Keep track of the full size of the hull.
    - Measurement Means: Just maintain variables that track these values in each function. Hull Size is already found in the algorithm
- Correctness Metrics
    - Ray Casting Algorithm: Perform this algorithm at the end of the program with the discovered hull points and iterate for each point in the input. If there are any outliers, the accuracy will be reduced
- Output File Inspection
    - The output file will be manually read and confirmed if it is written correctly, and if its information matches the output from the console.

Design Consideration

Programming Language

C++ will be chosen as it runs very quick and because the current developer (me) only knows C++ and Python. Additionally, C++ is a very powerful language with access to many of the characteristics in this algorithm, like the Object Oriented Programming, the time analysis methods, the different data structures, dynamic array space, etc.

The use of C++ will make the implementation a little harder as Python is an easier language to write in, but it is significantly faster than Python and will show much better execution times, especially with larger data sources.

Data Structure

Each of the data structures have been outlined in the algorithms, although they will be listed here again.

- Point Representation: Structs

    - Each implementation has a point struct within them that will be built for the dataset before execution of the rest of the functions. Some functions, like the orientation algorithms, rely on the struct format.
- All Point Storage: Dynamic Array
    - Each point has to be in sequence for the looping and indexing to work. This is especially important in Graham Scan, where sorting is performed on the same array.
    - We do not have a concrete maximum number of points that can be hard-coded. As such, it has to be dynamic.
- Hull Point Storage: Dynamic Array
    - Same reasoning as the previous
- Stack (Graham Scan): Array-based Stack Class
    - The Stack is used as pushing and popping from it is an O(1) operation. As it is array based, the peek operations will also be O(1).

File Format

CSV is the determined file format for the input and the output as the comma separation is very convenient and easy to read by the system. Since each point only has two dimensions and each point is in a new line, it is also very human-readable.

Console display will also be done for the output and the errors. Errors won't be saved into the output files.

Test Plan

Test Case	Input Size	Output Hull Size	Error	Faster Algorithm
No Input File	N/A	N/A	File Not Found	N/A
Empty File	0	N/A	Empty File	N/A
Invalid Data Format (non-numeric)	3	N/A	Invalid Data Format	N/A
Unreadable File	N/A	N/A	I/O Error	N/A
Incorrect Dimension Size (points without 2 coordinates)	3	N/A	Dimension Mismatch	N/A
Insufficient Points	2	N/A	Insufficient Points	N/A
Collinear Points, they should all have the same x-coordinate	≥3	N/A	Collinear Point	N/A
Triangle	3	3	N/A	Similar
Square	4	4	N/A	Similar

Triangle with duplicate points	6 (reduced to 3)	3	N/A	Similar
Random 10	10	?	N/A	Similar
Random 100	100	?	N/A	Graham Scan
Random 1000	1000	?	N/A	Graham Scan
Square with many points inside	1000	4	N/A	Jarvis