Convex Hull using Graham Scan - GeeksforGeeks

Source: https://www.geeksforgeeks.org/convex-hull-set-2-graham-scan/

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Convex Hull using Graham Scan Last Updated : 23 Jul, 2025 A convex hull is the smallest convex polygon that contains a given set of points. It is a useful concept in computational geometry and has applications in various fields such as computer graphics, image processing, and collision detection . A convex polygon is a polygon in which all interior angles are less than 180degrees . A convex hull can be constructed for any set of points, regardless of their arrangement. Convex Hull Examples Input: points[][] = [[0, 0], [1, -4], [-1, -5], [-5, -3], [-3, -1], [-1, -3], [-2, -2], [-1, -1], [-2, -1], [-1, 1]] Output: [[-5, -3], [-1, 1], [0, 0], [1, -4], [-1, -5] Explantation: The figure below shows the points of a convex polygon. These points define the boundary of the polygon . Approach: Prerequisite : How to check if two given line segments intersect? The Graham scan algorithm is a simple and efficient algorithm for computing the convex hull of a set of points. It works by iteratively adding points to the convex hull until all points have been added. The algorithm starts by finding the point with the smallest y-coordinate. This point is always on the convex hull. The algorithm then sorts the remaining points by their polar angle with respect to the starting point. The algorithm then iteratively adds points to the convex hull . At each step, the algorithm checks whether the last two points added to the convex hull form a right turn. If they do, then the last point is removed from the convex hull. Otherwise, the next point in the sorted list is added to the convex hull. Step by Step Approach Phase 1 (Sort points): The first step of the Graham Scan algorithm is to sort the points by their polar angle relative to the starting point. After sorting, the starting point is added to the convex hull, and the sorted points form a simple closed path. Phase 2 (Accept or Reject Points): After forming the closed path, we traverse it to remove concave points. Using orientation, we keep the first two points and check the next point by considering the last three points be prev(p) , curr(c) and next(n) . If the angle formed by these three points is not counterclockwise, we discard (reject) the current point, otherwise, we keep (accept) it. C++ #include <bits/stdc++.h> using namespace std ; // Structure to represent a point struct Point { double x , y ; // Operator to check equality of two points bool operator == (const Point & t) const { return x == t . x && y == t . y ; } }; // Function to find orientation of the triplet (a, b, c) // Returns -1 if clockwise, 1 if counter-clockwise, 0 if collinear int orientation (Point a , Point b , Point c) { double v = a . x * (b . y - c . y) + b . x * (c . y - a . y) + c . x * (a . y - b . y); if (v < 0) return -1 ; if (v > 0) return + 1 ; return 0 ; } // Function to calculate the squared distance between two points double distSq (Point a , Point b) { return (a . x - b . x) * (a . x - b . x) + (a . y - b . y) * (a . y - b . y); } // Function to find the convex hull of a set of 2D points vector < vector < int >> findConvexHull (vector < vector < int >> points) { // Store number of points points int n = points . size (); // Convex hull is not possible if there are fewer than 3 points if (n < 3) return {{ -1 }}; // Convert points 2D vector into vector of Point structures vector < Point > a ; for (auto & p : points) { a . push_back ({(double) p [0], (double) p [1]}); } // Find the point with the lowest y-coordinate (and leftmost in case of tie) Point p0 = * min_element (a . begin (), a . end (), [](Point a , Point b) { return make_pair (a . y , a . x) < make_pair (b . y , b . x); }); // Sort points based on polar angle with respect to the reference point p0 sort (a . begin (), a . end (), [& p0](const Point & a , const Point & b) { int o = orientation (p0 , a , b); // If points are collinear, keep the farthest one last if (o == 0) { return distSq (p0 , a) < distSq (p0 , b); } // Otherwise, sort by counter-clockwise order return o < 0 ; }); // Vector to store the points on the convex hull vector < Point > st ; // Process each point to build the hull for (int i = 0 ; i < (int) a . size (); ++ i) { // While last two points and current point make a non-left turn, remove the middle one while (st . size () > 1 && orientation (st [st . size () - 2], st . back (), a [i]) >= 0) st . pop_back (); // Add the current point

to the hull st . push_back (a [i]); } // If fewer than 3 points in the final hull, return {-1} if (st . size () < 3) return {{ -1 }}; // Convert the final hull into a vector of vectors of integers vector < vector < int >> result ; for (auto & p : st) { result . push_back ({(int) p . x , (int) p . y }); } return result ; } int main () { // Define the points set of 2D points vector < vector < int >> points = { { 0 , 0 }, { 1 , -4 }, { -1 , -5 }, { -5 , -3 }, { -3 , -1 }, { -1 , -3 }, { -2 , -2 }, { -1 , -1 }, { -2 , -1 }, { -1 , 1 } }; // Call the function to compute the convex hull vector < vector < int >> hull = findConvexHull (points); // If hull contains only {-1}, print the error result if (hull . size () == 1 && hull [0]. size () == 1){ cout << hull [0][0] << " " ; } else { // Print each point on the convex hull for (auto & point : hull) { cout << point [0] << ", " << point [1] << " \n " ; } } return 0 ; } Java import java.util.* ; class GfG { // Class to represent a point with x and y coordinates static class Point { double x , y ; // Constructor to initialize point Point (double x , double y) { this . x = x ; this . y = y ; } // Override equals to compare two points @Override public boolean equals (Object obj) { if (this == obj) return true ; if (obj == null || getClass () != obj . getClass ()) return false ; Point t = (Point) obj ; return Double . compare (t . x , x) == 0 && Double . compare (t . y , y) == 0 ; } } // Function to calculate orientation of ordered triplet (a, b, c) static int orientation (Point a , Point b , Point c) { // Compute the cross product value double v = a . x * (b . y - c . y) + b . x * (c . y - a . y) + c . x * (a . y - b . y); // Return -1 for clockwise, 1 for counter-clockwise, 0 for collinear if (v < 0) return - 1 ; if (v > 0) return 1 ; return 0 ; } // Function to compute square of distance between two points static double distSq (Point a , Point b) { return (a . x - b . x) * (a . x - b . x) + (a . y - b . y) * (a . y - b . y); } // Function to find the convex hull of a set of points static int [][] findConvexHull (int [][] points) { // Get number of points points int n = points . length ; // Convex hull is not possible with less than 3 points if (n < 3) return new int [][] {{ - 1 }}; // Convert int[][] points to list of Point objects ArrayList < Point > a = new ArrayList <> (); for (int [] p : points) { a . add (new Point (p [0] , p [1])); } // Find the bottom-most point (and left-most if tie) Point p0 = Collections . min (a , (p1 , p2) -> { if (p1 . y != p2 . y) return Double . compare (p1 . y , p2 . y); return Double . compare (p1 . x , p2 . x); }); // Sort points based on polar angle with respect to p0 a . sort ((p1 , p2) -> { int o = orientation (p0 , p1 , p2); // If collinear, sort by distance from p0 if (o == 0) { return Double . compare (distSq (p0 , p1), distSq (p0 , p2)); } // Otherwise sort by orientation return (o < 0) ? - 1 : 1 ; }); // Stack to store the points of convex hull Stack < Point > st = new Stack <> (); // Traverse sorted points for (Point p : a) { // Remove last point while the angle formed is not counter-clockwise while (st . size () > 1 && orientation (st . get (st . size () - 2), st . peek (), p) >= 0) st . pop (); // Add current point to the convex hull st . push (p); } // If convex hull has less than 3 points, it's invalid if (st . size () < 3) return new int [][] {{ - 1 }}; // Convert the convex hull points into int[][] int [][] result = new int [st . size ()][2] ; int i = 0 ; for (Point p : st) { result [i][0] = (int) p . x ; result [i][1] = (int) p . y ; i ++ ; } return result ; } public static void main (String [] args) { // points set of points int [][] points = { { 0 , 0 }, { 1 , - 4 }, { - 1 , - 5 }, { - 5 , - 3 }, { - 3 , - 1 }, { - 1 , - 3 }, { - 2 , - 2 }, { - 1 , - 1 }, { - 2 , - 1 }, { - 1 , 1 } }; // Call function to get convex hull int [][] hull = findConvexHull (points); // Print result if (hull . length == 1 && hull [0] . length == 1) { System . out . println (hull [0][0]); } else { for (int [] point : hull) { System . out . println (point [0] + ", " + point [1]); } } } } Python import math from functools import cmp_to_key # Class to represent a point class Point : def __init__ (self , x , y): self . x = x self . y = y # Method to check equality of two points def __eq__ (self , other): return self . x == other . x and self . y == other . y # Function to find orientation of the triplet (a, b, c) # Returns -1 if clockwise, 1 if counter-clockwise, 0 if collinear def orientation (a , b , c): val = (a . x * (b . y - c . y)) + \ (b . x * (c . y - a . y)) + \ (c . x * (a . y - b . y)) if val < 0 : return - 1 # Clockwise elif val > 0 : return 1 # Counter-clockwise return 0 # Collinear # Function to calculate the squared distance between two points def distSq (a , b): return (a . x - b . x) ** 2 + (a . y - b . y) ** 2 # Function to find the convex hull from a list of 2D points def findConvexHull (points): n = len (points) # Convex hull is not possible if there are fewer than 3 points if n < 3 : return [[- 1]] # Convert list of coordinates to Point objects a = [Point (p [0], p [1]) for p in points] # Find the point with the lowest y-coordinate (and leftmost in case of tie) p0 = min (a , key = lambda p : (p . y , p . x)) # Sort points based on polar angle with p0 as reference def compare (p1 , p2): o = orientation (p0 , p1 , p2) if o == 0 : return distSq (p0 , p1) - distSq (p0 , p2) return - 1 if o < 0 else 1 # Sort using custom comparator a_sorted = sorted (a , key = cmp_to_key (compare)) # Remove collinear points (keep farthest) m = 1 for i in range (1 , len (a_sorted)): while i < len (a_sorted) - 1 and \ orientation (p0 , a_sorted [i], a_sorted [i + 1]) == 0 : i += 1 a_sorted [m] = a_sorted [i] m += 1 # Convex hull is not possible with fewer than 3 unique points if m < 3 : return [[- 1]] # Initialize stack with first two points st = [a_sorted [0], a_sorted [1]] # Process the remaining points for i in range (2 , m): while len (st) > 1 and \ orientation (st [- 2], st [- 1], a_sorted [i]) >= 0 : st . pop () st . append (a_sorted [i]) # Final check for valid hull if len (st) < 3 : return [[- 1]] # Convert points back to list of [x, y] return [[int (p . x), int (p . y)] for p in st] # Test case

points = [[0 , 0], [1 , - 4], [- 1 , - 5], [- 5 , - 3], [- 3 , - 1], [- 1 , - 3], [- 2 , - 2], [- 1 , - 1], [- 2 , - 1], [- 1 , 1]] # Compute the convex hull hull = findConvexHull (points) # Output the result if len (hull) == 1 and hull [0][0] == - 1 : print (- 1) else : for point in hull : print (f " { point [0] } , { point [1] } ") C# using System ; using System.Collections.Generic ; class GfG { // Structure to represent a point struct Point { public double x , y ; public Point (double x , double y){ this . x = x ; this . y = y ; } // Function to compare two points public bool Equals (Point other){ return this . x == other . x && this . y == other . y ; } } // Function to find orientation of the triplet (a, b, c) // Returns -1 if clockwise, 1 if counter-clockwise, 0 if collinear static int orientation (Point a , Point b , Point c){ double v = a . x * (b . y - c . y) + b . x * (c . y - a . y) + c . x * (a . y - b . y); if (v < 0) return - 1 ; if (v > 0) return 1 ; return 0 ; } // Function to calculate the squared distance between two points static double distSq (Point a , Point b){ return (a . x - b . x) * (a . x - b . x) + (a . y - b . y) * (a . y - b . y); } // Function to find the convex hull of a set of 2D points static List < int [] > findConvexHull (int [,] input){ int n = input . GetLength (0); // Convex hull is not possible if there are fewer than 3 points if (n < 3) return new List < int [] > { new int [] { - 1 } }; // Convert input array into list of Point structs List < Point > a = new List < Point > (); for (int i = 0 ; i < n ; i ++){ a . Add (new Point (input [i , 0], input [i , 1])); } // Find the point with the lowest y-coordinate (and leftmost in case of tie) Point p0 = a [0]; foreach (var p in a) { if (p . y < p0 . y || (p . y == p0 . y && p . x < p0 . x)) { p0 = p ; } } // Sort points based on polar angle with respect to the reference point p0 a . Sort ((a1 , b1) => { int o = Orientation (p0 , a1 , b1); if (o == 0){ return DistSq (p0 , a1). CompareTo (DistSq (p0 , b1)); } return o < 0 ? - 1 : 1 ; }); // List to store the points on the convex hull List < Point > st = new List < Point > (); // Process each point to build the hull foreach (var pt in a){ // While last two points and current point make a non-left turn, remove the middle one while (st . Count > 1 && Orientation (st [st . Count - 2], st [st . Count - 1], pt) >= 0) { st . RemoveAt (st . Count - 1); } // Add the current point to the hull st . Add (pt); } // If fewer than 3 points in the final hull, return [-1] if (st . Count < 3) return new List < int [] > { new int [] { - 1 } }; // Convert the final hull into List<int[]> List < int [] > result = new List < int [] > (); foreach (var p in st) { result . Add (new int [] { (int) p . x , (int) p . y }); } return result ; } static void Main () { // Define the input set of 2D points int [,] points = new int [,] { { 0 , 0 }, { 1 , - 4 }, { - 1 , - 5 }, { - 5 , - 3 }, { - 3 , - 1 }, { - 1 , - 3 }, { - 2 , - 2 }, { - 1 , - 1 }, { - 2 , - 1 }, { - 1 , 1 } }; // Call the function to compute the convex hull List < int [] > hull = FindConvexHull (points); // If hull contains only [-1], print the error result if (hull . Count == 1 && hull [0]. Length == 1){ Console . WriteLine (hull [0][0]); } else { // Print each point on the convex hull foreach (var point in hull){ Console . WriteLine (point [0] + ", " + point [1]); } } } } JavaScript // Class to represent a point class Point { constructor (x , y) { this . x = x ; this . y = y ; } // Method to check equality of two points equals (t) { return this . x === t . x && this . y === t . y ; } } // Function to compute orientation of the triplet (a, b, c) // Returns -1 for clockwise, 1 for counter-clockwise, 0 for collinear function orientation (a , b , c) { const v = a . x * (b . y - c . y) + b . x * (c . y - a . y) + c . x * (a . y - b . y); if (v < 0) return - 1 ; // clockwise if (v > 0) return + 1 ; // counter-clockwise return 0 ; // collinear } // Function to compute squared distance between two points function distSq (a , b) { return (a . x - b . x) * (a . x - b . x) + (a . y - b . y) * (a . y - b . y); } // Function to find the convex hull of a set of points function findConvexHull (points) { const n = points . length ; // Convex hull is not possible if there are fewer than 3 points if (n < 3) return [[- 1]]; // Convert input array to Point objects const a = points . map (p => new Point (p [0], p [1])); // Find the point with the lowest y-coordinate (and leftmost if tie) const p0 = a . reduce ((min , p) => (p . y < min . y || (p . y === min . y && p . x < min . x)) ? p : min , a [0]); // Sort the points by polar angle with respect to p0 a . sort ((a , b) => { const o = orientation (p0 , a , b); // If collinear, place the farther point later if (o === 0) { return distSq (p0 , a) - distSq (p0 , b); } // Otherwise, order based on counter-clockwise direction return o < 0 ? - 1 : 1 ; }); // Remove duplicate collinear points (keep farthest one) let m = 1 ; for (let i = 1 ; i < a . length ; i ++) { // Skip closer collinear points while (i < a . length - 1 && orientation (p0 , a [i], a [i + 1]) === 0) { i ++ ; } // Keep current point in place a [m] = a [i]; m ++ ; } // If fewer than 3 points remain, hull is not possible if (m < 3) return [[- 1]]; // Initialize the convex hull stack with first two points const st = [a [0], a [1]]; // Process the remaining points for (let i = 2 ; i < m ; i ++) { // While the last three points do not make a left turn, pop the middle one while (st . length > 1 && orientation (st [st . length - 2], st [st . length - 1], a [i]) >= 0) { st . pop (); } // Add current point to stack st . push (a [i]); } // Final validation: if fewer than 3 points in stack, hull is not valid if (st . length < 3) return [[- 1]]; // Convert hull points to [x, y] arrays return st . map (p => [Math . round (p . x), Math . round (p . y)]); } // Test case const points = [[0 , 0], [1 , - 4], [- 1 , - 5], [- 5 , - 3], [- 3 , - 1], [- 1 , - 3], [- 2 , - 2], [- 1 , - 1], [- 2 , - 1], [- 1 , 1]]; // Compute the convex hull const hull = findConvexHull (points); // Output the result if (hull . length === 1 && hull [0][0] === - 1) { console . log (- 1); } else { hull . forEach (point => { console . log (` ${ point [ 0 ] } , ${ point [ 1 ] } `); }); } Output -1 -5 1 -4 0 0 -3 -1

-5 -3 Time Complexity: O(n log n), for finding the bottom-most point takes O(n), sorting the points takes O(n log n), and building the hull through stack operations takes O(n). Space Complexity: O(n), due to the stack used for storing the points during the hull construction, with no significant additional space required. Applications of Convex Hulls: Convex hulls have a wide range of applications, including: Collision detection: Convex hulls can be used to quickly determine whether two objects are colliding. This is useful in computer graphics and physics simulations. Image processing: Convex hulls can be used to segment objects in images. This is useful for tasks such as object recognition and tracking. Computational geometry: Convex hulls are used in a variety of computational geometry algorithms, such as finding the closest pair of points and computing the diameter of a set of points. Comment Article Tags: Article Tags: Mathematical Geometric DSA Morgan Stanley Samsung + 1 More