

Contents

1	Closest Pair of Points $O(n \log n)$ Implementation	2
	Source	6
2	Find if two rectangles overlap	7
	Source	8
3	Check whether a given point lies inside a triangle or not	9
	Source	10
4	How to check if a given point lies inside or outside a polygon?	12
	Source	16
5	How to check if two given line segments intersect?	17
	Source	20
6	Given n line segments, find if any two segments intersect	21
	Source	24
7	Minimum Cost Polygon Triangulation	25
	Source	29
8	Convex Hull Set 1 (Jarvis's Algorithm or Wrapping)	30
	Source	32
9	Convex Hull Set 2 (Graham Scan)	33
	Source	37

Chapter 1

Closest Pair of Points | $O(n \log n)$ Implementation

We are given an array of n points in the plane, and the problem is to find out the closest pair of points in the array. This problem arises in a number of applications. For example, in air-traffic control, you may want to monitor planes that come too close together, since this may indicate a possible collision. Recall the following formula for distance between two points p and q .

$$\|pq\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}.$$

We have discussed a [divide and conquer solution](#) for this problem. The time complexity of the implementation provided in the previous post is $O(n (\log n)^2)$. In this post, we discuss an implementation with time complexity as $O(n \log n)$.

Following is a recap of the algorithm discussed in the previous post.

- 1) We sort all points according to x coordinates.
- 2) Divide all points in two halves.
- 3) Recursively find the smallest distances in both subarrays.
- 4) Take the minimum of two smallest distances. Let the minimum be d .
- 5) Create an array `strip[]` that stores all points which are at most d distance away from the middle line dividing the two sets.
- 6) Find the smallest distance in `strip[]`.
- 7) Return the minimum of d and the smallest distance calculated in above step 6.

The great thing about the above approach is, if the array `strip[]` is sorted according to y coordinate, then we can find the smallest distance in `strip[]` in $O(n)$ time. In the implementation discussed in previous post, `strip[]` was explicitly sorted in every recursive call that made the time complexity $O(n (\log n)^2)$, assuming that the sorting step takes $O(n \log n)$ time.

In this post, we discuss an implementation where the time complexity is $O(n \log n)$. The idea is to presort all points according to y coordinates. Let the sorted array be `Py[]`. When we make recursive calls, we need to divide points of `Py[]` also according to the vertical line. We can do that by simply processing every point and comparing its x coordinate with x coordinate of middle line.

Following is C++ implementation of $O(n \log n)$ approach.

```

// A divide and conquer program in C++ to find the smallest distance from a
// given set of points.

#include <iostream>
#include <float.h>
#include <stdlib.h>
#include <math.h>
using namespace std;

// A structure to represent a Point in 2D plane
struct Point
{
    int x, y;
};

/* Following two functions are needed for library function qsort().
Refer: http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ */

// Needed to sort array of points according to X coordinate
int compareX(const void* a, const void* b)
{
    Point *p1 = (Point *)a, *p2 = (Point *)b;
    return (p1->x - p2->x);
}

// Needed to sort array of points according to Y coordinate
int compareY(const void* a, const void* b)
{
    Point *p1 = (Point *)a, *p2 = (Point *)b;
    return (p1->y - p2->y);
}

// A utility function to find the distance between two points
float dist(Point p1, Point p2)
{
    return sqrt( (p1.x - p2.x)*(p1.x - p2.x) +
                (p1.y - p2.y)*(p1.y - p2.y)
                );
}

// A Brute Force method to return the smallest distance between two points
// in P[] of size n
float bruteForce(Point P[], int n)
{
    float min = FLT_MAX;
    for (int i = 0; i < n; ++i)
        for (int j = i+1; j < n; ++j)
            if (dist(P[i], P[j]) < min)
                min = dist(P[i], P[j]);
    return min;
}

// A utility function to find minimum of two float values

```

```

float min(float x, float y)
{
    return (x < y)? x : y;
}

// A utility function to find the distance between the closest points of
// strip of given size. All points in strip[] are sorted accordint to
// y coordinate. They all have an upper bound on minimum distance as d.
// Note that this method seems to be a  $O(n^2)$  method, but it's a  $O(n)$ 
// method as the inner loop runs at most 6 times
float stripClosest(Point strip[], int size, float d)
{
    float min = d; // Initialize the minimum distance as d

    // Pick all points one by one and try the next points till the difference
    // between y coordinates is smaller than d.
    // This is a proven fact that this loop runs at most 6 times
    for (int i = 0; i < size; ++i)
        for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)
            if (dist(strip[i],strip[j]) < min)
                min = dist(strip[i], strip[j]);

    return min;
}

// A recursive function to find the smallest distance. The array Px contains
// all points sorted according to x coordinates and Py contains all points
// sorted according to y coordinates
float closestUtil(Point Px[], Point Py[], int n)
{
    // If there are 2 or 3 points, then use brute force
    if (n <= 3)
        return bruteForce(Px, n);

    // Find the middle point
    int mid = n/2;
    Point midPoint = Px[mid];

    // Divide points in y sorted array around the vertical line.
    // Assumption: All x coordinates are distinct.
    Point Pyl[mid+1]; // y sorted points on left of vertical line
    Point Pyr[n-mid-1]; // y sorted points on right of vertical line
    int li = 0, ri = 0; // indexes of left and right subarrays
    for (int i = 0; i < n; i++)
    {
        if (Py[i].x <= midPoint.x)
            Pyl[li++] = Py[i];
        else
            Pyr[ri++] = Py[i];
    }

    // Consider the vertical line passing through the middle point

```

```

    // calculate the smallest distance dl on left of middle point and
    // dr on right side
    float dl = closestUtil(Px, Py1, mid);
    float dr = closestUtil(Px + mid, Pyr, n-mid);

    // Find the smaller of two distances
    float d = min(dl, dr);

    // Build an array strip[] that contains points close (closer than d)
    // to the line passing through the middle point
    Point strip[n];
    int j = 0;
    for (int i = 0; i < n; i++)
        if (abs(Py[i].x - midPoint.x) < d)
            strip[j] = Py[i], j++;

    // Find the closest points in strip. Return the minimum of d and closest
    // distance is strip[]
    return min(d, stripClosest(strip, j, d) );
}

// The main functin that finds the smallest distance
// This method mainly uses closestUtil()
float closest(Point P[], int n)
{
    Point Px[n];
    Point Py[n];
    for (int i = 0; i < n; i++)
    {
        Px[i] = P[i];
        Py[i] = P[i];
    }

    qsort(Px, n, sizeof(Point), compareX);
    qsort(Py, n, sizeof(Point), compareY);

    // Use recursive function closestUtil() to find the smallest distance
    return closestUtil(Px, Py, n);
}

// Driver program to test above functions
int main()
{
    Point P[] = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}};
    int n = sizeof(P) / sizeof(P[0]);
    cout << "The smallest distance is " << closest(P, n);
    return 0;
}

```

Output:

The smallest distance is 1.41421

Time Complexity: Let Time complexity of above algorithm be $T(n)$. Let us assume that we use a $O(n \log n)$ sorting algorithm. The above algorithm divides all points in two sets and recursively calls for two sets. After dividing, it finds the strip in $O(n)$ time. Also, it takes $O(n)$ time to divide the P_y array around the mid vertical line. Finally finds the closest points in strip in $O(n)$ time. So $T(n)$ can be expressed as follows

$$T(n) = 2T(n/2) + O(n) + O(n) + O(n)$$

$$T(n) = 2T(n/2) + O(n)$$

$$T(n) = O(n \log n)$$

References:

<http://www.cs.umd.edu/class/fall2013/cmsc451/Lects/lect10.pdf>

<http://www.youtube.com/watch?v=vS4Zn1a9KUc>

<http://www.youtube.com/watch?v=T3T7T8Ym20M>

http://en.wikipedia.org/wiki/Closest_pair_of_points_problem

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<http://www.geeksforgeeks.org/closest-pair-of-points-onlogn-implementation/>

Category: [Misc](#) Tags: [Divide and Conquer](#), [geometric algorithms](#)

Post navigation

[← Amazon Interview | Set 54 \(On Campus for SDE\) Deepest left leaf node in a binary tree](#) [→](#)

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 2

Find if two rectangles overlap

Given two rectangles, find if the given two rectangles overlap or not.

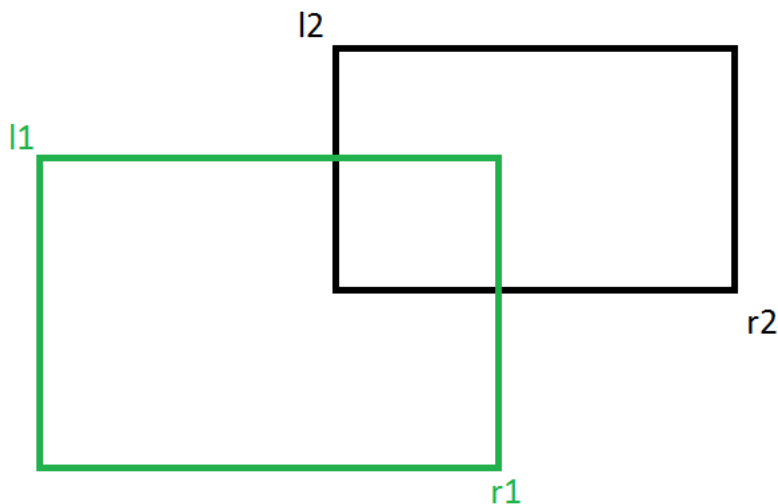
Note that a rectangle can be represented by two coordinates, top left and bottom right. So mainly we are given following four coordinates.

l1: Top Left coordinate of first rectangle.

r1: Bottom Right coordinate of first rectangle.

l2: Top Left coordinate of second rectangle.

r2: Bottom Right coordinate of second rectangle.



We need to write a function *bool doOverlap(l1, r1, l2, r2)* that returns true if the two given rectangles overlap.

One solution is to one by one pick all points of one rectangle and [see if the point lies inside the other rectangle or not](#). This can be done using the algorithm discussed [here](#).

Following is a simpler approach. Two rectangles **do not** overlap if one of the following conditions is true.

- 1) One rectangle is above top edge of other rectangle.
- 2) One rectangle is on left side of left edge of other rectangle.

We need to check above cases to find out if given rectangles overlap or not. Following is C++ implementation of the above approach.

```
#include<stdio.h>
```

```

struct Point
{
    int x, y;
};

// Returns true if two rectangles (l1, r1) and (l2, r2) overlap
bool doOverlap(Point l1, Point r1, Point l2, Point r2)
{
    // If one rectangle is on left side of other
    if (l1.x > r2.x || l2.x > r1.x)
        return false;

    // If one rectangle is above other
    if (l1.y < r2.y || l2.y < r1.y)
        return false;

    return true;
}

/* Driver program to test above function */
int main()
{
    Point l1 = {0, 10}, r1 = {10, 0};
    Point l2 = {5, 5}, r2 = {15, 0};
    if (doOverlap(l1, r1, l2, r2))
        printf("Rectangles Overlap");
    else
        printf("Rectangles Don't Overlap");
    return 0;
}

```

Output:

Rectangles Overlap

Time Complexity of above code is $O(1)$ as the code doesn't have any loop or recursion.

This article is compiled by **Aman Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<http://www.geeksforgeeks.org/find-two-rectangles-overlap/>

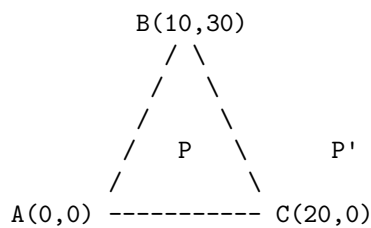
Category: Misc Tags: [geometric algorithms](#)

Chapter 3

Check whether a given point lies inside a triangle or not

Given three corner points of a triangle, and one more point P. Write a function to check whether P lies within the triangle or not.

For example, consider the following program, the function should return true for P(10, 15) and false for P'(30, 15)



Source: [Microsoft Interview Question](#)

Solution:

Let the coordinates of three corners be (x1, y1), (x2, y2) and (x3, y3). And coordinates of the given point P be (x, y)

- 1) Calculate area of the given triangle, i.e., area of the triangle ABC in the above diagram. Area A = $[x1(y2 - y3) + x2(y3 - y1) + x3(y1 - y2)]/2$
- 2) Calculate area of the triangle PAB. We can use the same formula for this. Let this area be A1.
- 3) Calculate area of the triangle PBC. Let this area be A2.
- 4) Calculate area of the triangle PAC. Let this area be A3.
- 5) If P lies inside the triangle, then A1 + A2 + A3 must be equal to A.

```
#include <stdio.h>
#include <stdlib.h>

/* A utility function to calculate area of triangle formed by (x1, y1),
   (x2, y2) and (x3, y3) */
float area(int x1, int y1, int x2, int y2, int x3, int y3)
```

```

{
    return abs((x1*(y2-y3) + x2*(y3-y1)+ x3*(y1-y2))/2.0);
}

/* A function to check whether point P(x, y) lies inside the triangle formed
   by A(x1, y1), B(x2, y2) and C(x3, y3) */
bool isInside(int x1, int y1, int x2, int y2, int x3, int y3, int x, int y)
{
    /* Calculate area of triangle ABC */
    float A = area (x1, y1, x2, y2, x3, y3);

    /* Calculate area of triangle PBC */
    float A1 = area (x, y, x2, y2, x3, y3);

    /* Calculate area of triangle PAC */
    float A2 = area (x1, y1, x, y, x3, y3);

    /* Calculate area of triangle PAB */
    float A3 = area (x1, y1, x2, y2, x, y);

    /* Check if sum of A1, A2 and A3 is same as A */
    return (A == A1 + A2 + A3);
}

/* Driver program to test above function */
int main()
{
    /* Let us check whether the point P(10, 15) lies inside the triangle
       formed by A(0, 0), B(20, 0) and C(10, 30) */
    if (isInside(0, 0, 20, 0, 10, 30, 10, 15))
        printf ("Inside");
    else
        printf ("Not Inside");

    return 0;
}

```

Ouptut:

Inside

Exercise: Given coordinates of four corners of a rectangle, and a point P. Write a function to check whether P lies inside the given rectangle or not.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

<http://www.geeksforgeeks.org/check-whether-a-given-point-lies-inside-a-triangle-or-not/>

Category: Misc Tags: MathematicalAlgo

Post navigation

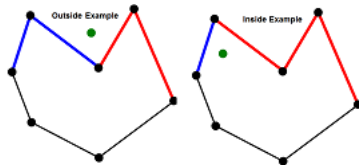
[← Copy elision in C++ Count numbers that don't contain 3](#) [→](#)

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 4

How to check if a given point lies inside or outside a polygon?

Given a polygon and a point 'p', find if 'p' lies inside the polygon or not. The points lying on the border are considered inside.

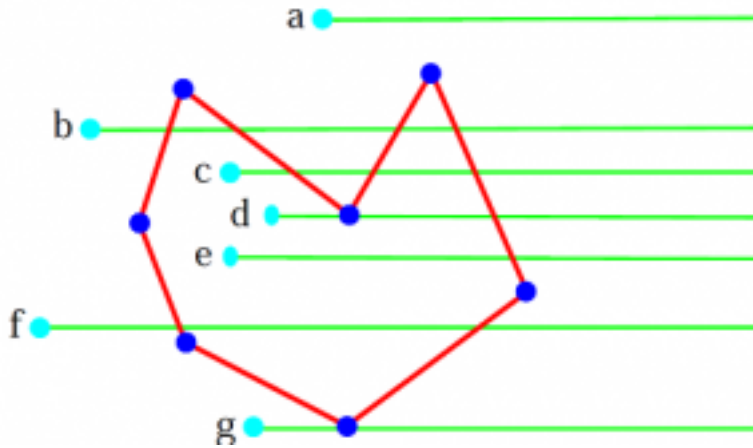


We strongly recommend to see the following post first.

[How to check if two given line segments intersect?](#)

Following is a simple idea to check whether a point is inside or outside.

- 1) Draw a horizontal line to the right of each point and extend it to infinity
- 1) Count the number of times the line intersects with polygon edges.
- 2) A point is inside the polygon if either count of intersections is odd or point lies on an edge of polygon. If none of the conditions is true, then point lies outside.



How to handle point 'g' in the above figure?

Note that we should return true if the point lies on the line or same as one of the vertices of the given polygon. To handle this, after checking if the line from 'p' to extreme intersects, we check whether 'p' is colinear with vertices of current line of polygon. If it is colinear, then we check if the point 'p' lies on current side of polygon, if it lies, we return true, else false.

Following is C++ implementation of the above idea.

```
// A C++ program to check if a given point lies inside a given polygon
// Refer http://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/
// for explanation of functions onSegment(), orientation() and doIntersect()
#include <iostream>
using namespace std;

// Define Infinite (Using INT_MAX caused overflow problems)
#define INF 10000

struct Point
{
    int x;
    int y;
};

// Given three colinear points p, q, r, the function checks if
// point q lies on line segment 'pr'
bool onSegment(Point p, Point q, Point r)
{
    if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&
        q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))
        return true;
    return false;
}

// To find orientation of ordered triplet (p, q, r).
// The function returns following values
// 0 --> p, q and r are colinear
// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(Point p, Point q, Point r)
```

```

{
    int val = (q.y - p.y) * (r.x - q.x) -
              (q.x - p.x) * (r.y - q.y);

    if (val == 0) return 0; // colinear
    return (val > 0)? 1: 2; // clock or counterclock wise
}

// The function that returns true if line segment 'p1q1'
// and 'p2q2' intersect.
bool doIntersect(Point p1, Point q1, Point p2, Point q2)
{
    // Find the four orientations needed for general and
    // special cases
    int o1 = orientation(p1, q1, p2);
    int o2 = orientation(p1, q1, q2);
    int o3 = orientation(p2, q2, p1);
    int o4 = orientation(p2, q2, q1);

    // General case
    if (o1 != o2 && o3 != o4)
        return true;

    // Special Cases
    // p1, q1 and p2 are colinear and p2 lies on segment p1q1
    if (o1 == 0 && onSegment(p1, p2, q1)) return true;

    // p1, q1 and p2 are colinear and q2 lies on segment p1q1
    if (o2 == 0 && onSegment(p1, q2, q1)) return true;

    // p2, q2 and p1 are colinear and p1 lies on segment p2q2
    if (o3 == 0 && onSegment(p2, p1, q2)) return true;

    // p2, q2 and q1 are colinear and q1 lies on segment p2q2
    if (o4 == 0 && onSegment(p2, q1, q2)) return true;

    return false; // Doesn't fall in any of the above cases
}

// Returns true if the point p lies inside the polygon[] with n vertices
bool isInside(Point polygon[], int n, Point p)
{
    // There must be at least 3 vertices in polygon[]
    if (n < 3) return false;

    // Create a point for line segment from p to infinite
    Point extreme = {INF, p.y};

    // Count intersections of the above line with sides of polygon
    int count = 0, i = 0;
    do
    {
        int next = (i+1)%n;

```

```

    // Check if the line segment from 'p' to 'extreme' intersects
    // with the line segment from 'polygon[i]' to 'polygon[next]'
    if (doIntersect(polygon[i], polygon[next], p, extreme))
    {
        // If the point 'p' is colinear with line segment 'i-next',
        // then check if it lies on segment. If it lies, return true,
        // otherwise false
        if (orientation(polygon[i], p, polygon[next]) == 0)
            return onSegment(polygon[i], p, polygon[next]);

        count++;
    }
    i = next;
} while (i != 0);

// Return true if count is odd, false otherwise
return count&1; // Same as (count%2 == 1)
}

// Driver program to test above functions
int main()
{
    Point polygon1[] = {{0, 0}, {10, 0}, {10, 10}, {0, 10}};
    int n = sizeof(polygon1)/sizeof(polygon1[0]);
    Point p = {20, 20};
    isInside(polygon1, n, p)? cout << "Yes \n": cout << "No \n";

    p = {5, 5};
    isInside(polygon1, n, p)? cout << "Yes \n": cout << "No \n";

    Point polygon2[] = {{0, 0}, {5, 5}, {5, 0}};
    p = {3, 3};
    n = sizeof(polygon2)/sizeof(polygon2[0]);
    isInside(polygon2, n, p)? cout << "Yes \n": cout << "No \n";

    p = {5, 1};
    isInside(polygon2, n, p)? cout << "Yes \n": cout << "No \n";

    p = {8, 1};
    isInside(polygon2, n, p)? cout << "Yes \n": cout << "No \n";

    Point polygon3[] = {{0, 0}, {10, 0}, {10, 10}, {0, 10}};
    p = {-1, 10};
    n = sizeof(polygon3)/sizeof(polygon3[0]);
    isInside(polygon3, n, p)? cout << "Yes \n": cout << "No \n";

    return 0;
}

```

Output:

No
Yes

Yes
Yes
No
No

Time Complexity: $O(n)$ where n is the number of vertices in the given polygon.

Source:

<http://www.dcs.gla.ac.uk/~pat/52233/slides/Geometry1x1.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<http://www.geeksforgeeks.org/how-to-check-if-a-given-point-lies-inside-a-polygon/>

Category: [Misc](#) Tags: [geometric algorithms](#), [MathematicalAlgo](#)

Post navigation

[← How to check if two given line segments intersect? Convex Hull | Set 1 \(Jarvis's Algorithm or Wrapping\)](#)
[→](#)

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 5

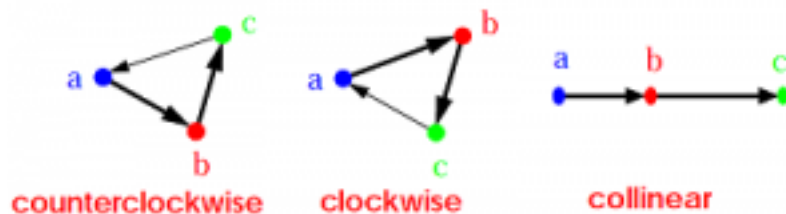
How to check if two given line segments intersect?

Given two line segments $(p1, q1)$ and $(p2, q2)$, find if the given line segments intersect with each other.

Before we discuss solution, let us define notion of **orientation**. Orientation of an ordered triplet of points in the plane can be

- counterclockwise
- clockwise
- collinear

The following diagram shows different possible orientations of (a, b, c)



Note the word 'ordered' here. Orientation of (a, b, c) may be different from orientation of (c, b, a) .

How is Orientation useful here?

Two segments $(p1, q1)$ and $(p2, q2)$ intersect if and only if one of the following two conditions is verified

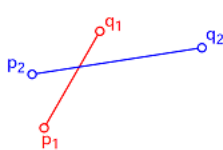
1. General Case:

- $(p1, q1, p2)$ and $(p1, q1, q2)$ have different orientations and
- $(p2, q2, p1)$ and $(p2, q2, q1)$ have different orientations

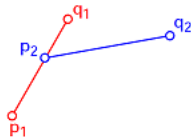
2. Special Case

- $(p1, q1, p2)$, $(p1, q1, q2)$, $(p2, q2, p1)$, and $(p2, q2, q1)$ are all collinear and
- the x-projections of $(p1, q1)$ and $(p2, q2)$ intersect
- the y-projections of $(p1, q1)$ and $(p2, q2)$ intersect

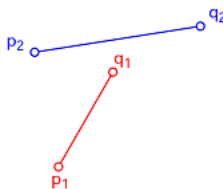
Examples of General Case:



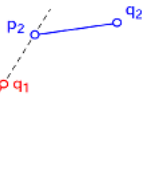
Example 1: Orientations of (p_1, q_1, p_2) and (p_1, q_1, q_2) are different. Orientations of (p_2, q_2, p_1) and (p_2, q_2, q_1) are also different



Example 2: Orientations of (p_1, q_1, p_2) and (p_1, q_1, q_2) are different. Orientations of (p_2, q_2, p_1) and (p_2, q_2, q_1) are also different



Example 3: Orientations of (p_1, q_1, p_2) and (p_1, q_1, q_2) are different. Orientations of (p_2, q_2, p_1) and (p_2, q_2, q_1) are same

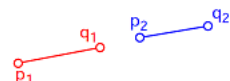


Example 4: Orientations of (p_1, q_1, p_2) and (p_1, q_1, q_2) are different. Orientations of (p_2, q_2, p_1) and (p_2, q_2, q_1) are same

Examples of Special Case:



Example 1: All points are collinear. The x-projections of (p_1, q_1) and (p_2, q_2) intersect. The y-projections of (p_1, q_1) and (p_2, q_2) intersect



Example 2: All points are collinear. The x-projections of (p_1, q_1) and (p_2, q_2) do not intersect. The y-projections of (p_1, q_1) and (p_2, q_2) intersect

Following is C++ implementation based on above idea.

```
// A C++ program to check if two given line segments intersect
#include <iostream>
using namespace std;

struct Point
{
    int x;
    int y;
};

// Given three collinear points p, q, r, the function checks if
// point q lies on line segment 'pr'
bool onSegment(Point p, Point q, Point r)
{
    if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&
        q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))
        return true;

    return false;
}

// To find orientation of ordered triplet (p, q, r).
// The function returns following values
// 0 --> p, q and r are collinear
// 1 --> Clockwise
// 2 --> Counterclockwise
```

```

int orientation(Point p, Point q, Point r)
{
    // See 10th slides from following link for derivation of the formula
    // http://www.dcs.gla.ac.uk/~pat/52233/slides/Geometry1x1.pdf
    int val = (q.y - p.y) * (r.x - q.x) -
              (q.x - p.x) * (r.y - q.y);

    if (val == 0) return 0; // colinear

    return (val > 0)? 1: 2; // clock or counterclock wise
}

// The main function that returns true if line segment 'p1q1'
// and 'p2q2' intersect.
bool doIntersect(Point p1, Point q1, Point p2, Point q2)
{
    // Find the four orientations needed for general and
    // special cases
    int o1 = orientation(p1, q1, p2);
    int o2 = orientation(p1, q1, q2);
    int o3 = orientation(p2, q2, p1);
    int o4 = orientation(p2, q2, q1);

    // General case
    if (o1 != o2 && o3 != o4)
        return true;

    // Special Cases
    // p1, q1 and p2 are colinear and p2 lies on segment p1q1
    if (o1 == 0 && onSegment(p1, p2, q1)) return true;

    // p1, q1 and p2 are colinear and q2 lies on segment p1q1
    if (o2 == 0 && onSegment(p1, q2, q1)) return true;

    // p2, q2 and p1 are colinear and p1 lies on segment p2q2
    if (o3 == 0 && onSegment(p2, p1, q2)) return true;

    // p2, q2 and q1 are colinear and q1 lies on segment p2q2
    if (o4 == 0 && onSegment(p2, q1, q2)) return true;

    return false; // Doesn't fall in any of the above cases
}

// Driver program to test above functions
int main()
{
    struct Point p1 = {1, 1}, q1 = {10, 1};
    struct Point p2 = {1, 2}, q2 = {10, 2};

    doIntersect(p1, q1, p2, q2)? cout << "Yes\n": cout << "No\n";

    p1 = {10, 0}, q1 = {0, 10};
    p2 = {0, 0}, q2 = {10, 10};
    doIntersect(p1, q1, p2, q2)? cout << "Yes\n": cout << "No\n";
}

```

```

    p1 = {-5, -5}, q1 = {0, 0};
    p2 = {1, 1}, q2 = {10, 10};
    doIntersect(p1, q1, p2, q2)? cout << "Yes\n": cout << "No\n";

    return 0;
}

```

Output:

```

No
Yes
No

```

Sources:

<http://www.dcs.gla.ac.uk/~pat/52233/slides/Geometry1x1.pdf>

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<http://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/>

Category: [Misc](#) Tags: [geometric algorithms](#), [MathematicalAlgo](#)

Post navigation

[← Dynamic Programming | Set 35 \(Longest Arithmetic Progression\)](#) [How to check if a given point lies inside or outside a polygon? →](#)

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 6

Given n line segments, find if any two segments intersect

We have discussed the problem to detect if [two given line segments intersect or not](#). In this post, we extend the problem. Here we are given n line segments and we need to find out if any two line segments intersect or not.

Naive Algorithm A naive solution to solve this problem is to check every pair of lines and check if the pair intersects or not. [We can check two line segments in \$O\(1\)\$ time](#). Therefore, this approach takes $O(n^2)$.

Sweep Line Algorithm: We can solve this problem in $O(n \log n)$ time using Sweep Line Algorithm. The algorithm first sorts the end points along the x axis from left to right, then it passes a vertical line through all points from left to right and checks for intersections. Following are detailed steps.

1) Let there be n given lines. There must be $2n$ end points to represent the n lines. Sort all points according to x coordinates. While sorting maintain a flag to indicate whether this point is left point of its line or right point.

2) Start from the leftmost point. Do following for every point

....a) If the current point is a left point of its line segment, check for intersection of its line segment with the segments just above and below it. And add its line to *active* line segments (line segments for which left end point is seen, but right end point is not seen yet). Note that we consider only those neighbors which are still active.

....b) If the current point is a right point, remove its line segment from active list and check whether its two active neighbors (points just above and below) intersect with each other.

The step 2 is like passing a vertical line from all points starting from the leftmost point to the rightmost point. That is why this algorithm is called Sweep Line Algorithm. The Sweep Line technique is useful in many other geometric algorithms like [calculating the 2D Voronoi diagram](#)

What data structures should be used for efficient implementation?

In step 2, we need to store all active line segments. We need to do following operations efficiently:

- Insert a new line segment
- Delete a line segment
- Find predecessor and successor according to y coordinate values

The obvious choice for above operations is Self-Balancing Binary Search Tree like AVL Tree, Red Black Tree. With a Self-Balancing BST, we can do all of the above operations in $O(\log n)$ time.

Also, in step 1, instead of sorting, we can use min heap data structure. Building a min heap takes $O(n)$ time and every extract min operation takes $O(\log n)$ time (See [this](#)).

PseudoCode:

The following pseudocode doesn't use heap. It simply sort the array.

```

sweepLineIntersection(Points[0..2n-1]):
1. Sort Points[] from left to right (according to x coordinate)

2. Create an empty Self-Balancing BST T. It will contain all active line
   Segments ordered by y coordinate.

// Process all 2n points
3. for i = 0 to 2n-1

    // If this point is left end of its line
    if (Points[i].isLeft)
        T.insert(Points[i].line()) // Insert into the tree

        // Check if this points intersects with its predecessor and successor
        if ( doIntersect(Points[i].line(), T.pred(Points[i].line()) )
            return true
        if ( doIntersect(Points[i].line(), T.succ(Points[i].line()) )
            return true

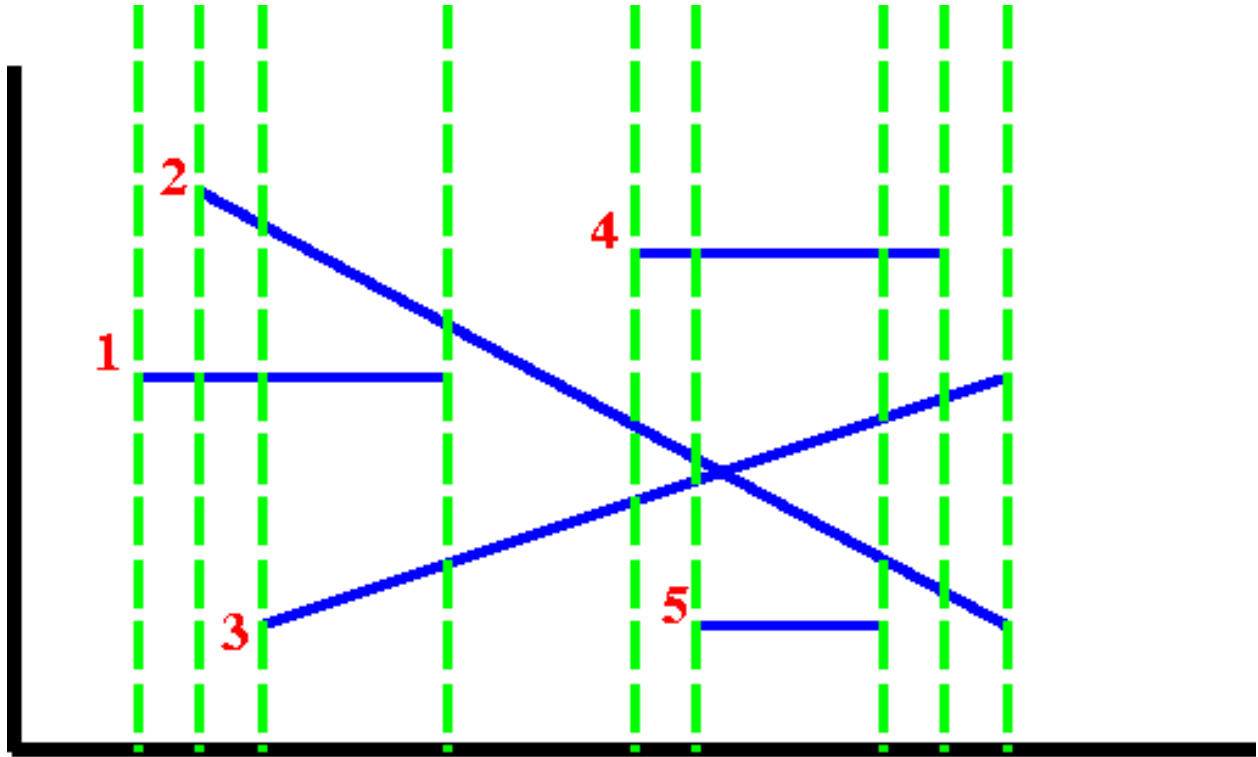
    else // If it's a right end of its line
        // Check if its predecessor and successor intersect with each other
        if ( doIntersect(T.pred(Points[i].line()), T.succ(Points[i].line()))
            return true
        T.delete(Points[i].line()) // Delete from tree

4. return False

```

Example:

Let us consider the following example taken from [here](#). There are 5 line segments 1, 2, 3, 4 and 5. The dotted green lines show sweep lines.



sweep lines

Following are steps followed by the algorithm. All points from left to right are processed one by one. We maintain a self-balancing binary search tree.

Left end point of line segment 1 is processed: 1 is inserted into the Tree. The tree contains 1. No intersection.

Left end point of line segment 2 is processed: Intersection of 1 and 2 is checked. 2 is inserted into the Tree. No intersection. The tree contains 1, 2.

Left end point of line segment 3 is processed: Intersection of 3 with 1 is checked. No intersection. 3 is inserted into the Tree. The tree contains 2, 1, 3.

Right end point of line segment 1 is processed: 1 is deleted from the Tree. Intersection of 2 and 3 is checked. Intersection of 2 and 3 is reported. The tree contains 2, 3. Note that the above pseudocode returns at this point. We can continue from here to report all intersection points.

Left end point of line segment 4 is processed: Intersections of line 4 with lines 2 and 3 are checked. No intersection. 4 is inserted into the Tree. The tree contains 2, 4, 3.

Left end point of line segment 5 is processed: Intersection of 5 with 3 is checked. No intersection. 5 is inserted into the Tree. The tree contains 2, 4, 3, 5.

Right end point of line segment 5 is processed: 5 is deleted from the Tree. The tree contains 2, 4, 3.

Right end point of line segment 4 is processed: 4 is deleted from the Tree. The tree contains 2, 4, 3. Intersection of 2 with 3 is checked. Intersection of 2 with 3 is reported. The tree contains 2, 3. Note that the intersection of 2 and 3 is reported again. We can add some logic to check for duplicates.

Right end point of line segment 2 and 3 are processed: Both are deleted from tree and tree becomes empty.

Time Complexity: The first step is sorting which takes $O(n \log n)$ time. The second step process $2n$ points and for processing every point, it takes $O(\log n)$ time. Therefore, overall time complexity is $O(n \log n)$

References:

<http://www.cs.uiuc.edu/~jeffe/teaching/373/notes/x06-sweepline.pdf>

<http://courses.csail.mit.edu/6.006/spring11/lectures/lec24.pdf>

<http://www.youtube.com/watch?v=dePDHVovJlE>

<http://www.eecs.wsu.edu/~cook/aa/lectures/125/node10.html>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

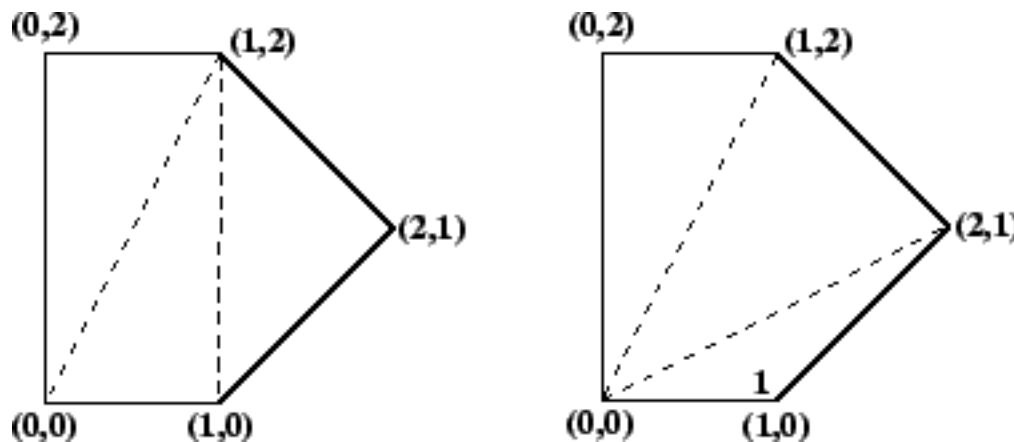
<http://www.geeksforgeeks.org/given-a-set-of-line-segments-find-if-any-two-segments-intersect/>

Chapter 7

Minimum Cost Polygon Triangulation

A triangulation of a convex polygon is formed by drawing diagonals between non-adjacent vertices (corners) such that the diagonals never intersect. The problem is to find the cost of triangulation with the minimum cost. The cost of a triangulation is sum of the weights of its component triangles. Weight of each triangle is its perimeter (sum of lengths of all sides)

See following example taken from [this](#) source.



Two triangulations of the same convex pentagon. The triangulation on the left has a cost of $8 + 2\sqrt{2} + 2\sqrt{5}$ (approximately 15.30), the one on the right has a cost of $4 + 2\sqrt{2} + 4\sqrt{5}$ (approximately 15.77).

This problem has recursive substructure. The idea is to divide the polygon into three parts: a single triangle, the sub-polygon to the left, and the sub-polygon to the right. We try all possible divisions like this and find the one that minimizes the cost of the triangle plus the cost of the triangulation of the two sub-polygons.

Let Minimum Cost of triangulation of vertices from i to j be $\text{minCost}(i, j)$

If j

Following is C++ implementation of above naive recursive formula.

```
// Recursive implementation for minimum cost convex polygon triangulation
#include <iostream>
#include <cmath>
#define MAX 1000000.0
using namespace std;
```

```

// Structure of a point in 2D plane
struct Point
{
    int x, y;
};

// Utility function to find minimum of two double values
double min(double x, double y)
{
    return (x <= y)? x : y;
}

// A utility function to find distance between two points in a plane
double dist(Point p1, Point p2)
{
    return sqrt((p1.x - p2.x)*(p1.x - p2.x) +
                (p1.y - p2.y)*(p1.y - p2.y));
}

// A utility function to find cost of a triangle. The cost is considered
// as perimeter (sum of lengths of all edges) of the triangle
double cost(Point points[], int i, int j, int k)
{
    Point p1 = points[i], p2 = points[j], p3 = points[k];
    return dist(p1, p2) + dist(p2, p3) + dist(p3, p1);
}

// A recursive function to find minimum cost of polygon triangulation
// The polygon is represented by points[i..j].
double mTC(Point points[], int i, int j)
{
    // There must be at least three points between i and j
    // (including i and j)
    if (j < i+2)
        return 0;

    // Initialize result as infinite
    double res = MAX;

    // Find minimum triangulation by considering all
    for (int k=i+1; k<j; k++)
        res = min(res, (mTC(points, i, k) + mTC(points, k, j) +
                        cost(points, i, k, j)));

    return res;
}

// Driver program to test above functions
int main()
{
    Point points[] = {{0, 0}, {1, 0}, {2, 1}, {1, 2}, {0, 2}};
    int n = sizeof(points)/sizeof(points[0]);
    cout << mTC(points, 0, n-1);
    return 0;
}

```

15.3006

[illegible]

It can be easily seen in the above recursion tree that the problem has many overlapping subproblems. Since the problem has both properties: **Optimal Substructure** and **Overlapping Subproblems**, it can be efficiently solved using dynamic programming.

```
// A Dynamic Programming based program to find minimum cost of convex
// polygon triangulation
#include <iostream>
#include <cmath>
#define MAX 1000000.0
using namespace std;

// Structure of a point in 2D plane
struct Point
{
    int x, y;
};

// Utility function to find minimum of two double values
double min(double x, double y)
{
    return (x <= y)? x : y;
}

// A utility function to find distance between two points in a plane
double dist(Point p1, Point p2)
{
    return sqrt((p1.x - p2.x)*(p1.x - p2.x) +
```

```

        (p1.y - p2.y)*(p1.y - p2.y));
    }

    // A utility function to find cost of a triangle. The cost is considered
    // as perimeter (sum of lengths of all edges) of the triangle
    double cost(Point points[], int i, int j, int k)
    {
        Point p1 = points[i], p2 = points[j], p3 = points[k];
        return dist(p1, p2) + dist(p2, p3) + dist(p3, p1);
    }

    // A Dynamic programming based function to find minimum cost for convex
    // polygon triangulation.
    double mTCDP(Point points[], int n)
    {
        // There must be at least 3 points to form a triangle
        if (n < 3)
            return 0;

        // table to store results of subproblems. table[i][j] stores cost of
        // triangulation of points from i to j. The entry table[0][n-1] stores
        // the final result.
        double table[n][n];

        // Fill table using above recursive formula. Note that the table
        // is filled in diagonal fashion i.e., from diagonal elements to
        // table[0][n-1] which is the result.
        for (int gap = 0; gap < n; gap++)
        {
            for (int i = 0, j = gap; j < n; i++, j++)
            {
                if (j < i+2)
                    table[i][j] = 0.0;
                else
                {
                    table[i][j] = MAX;
                    for (int k = i+1; k < j; k++)
                    {
                        double val = table[i][k] + table[k][j] + cost(points, i, j, k);
                        if (table[i][j] > val)
                            table[i][j] = val;
                    }
                }
            }
        }
        return table[0][n-1];
    }

    // Driver program to test above functions
    int main()
    {
        Point points[] = {{0, 0}, {1, 0}, {2, 1}, {1, 2}, {0, 2}};
        int n = sizeof(points)/sizeof(points[0]);
        cout << mTCDP(points, n);
    }

```

```
    return 0;
}
```

Output:

15.3006

Time complexity of the above dynamic programming solution is $O(n^3)$.

Please note that the above implementations assume that the points of convex polygon are given in order (either clockwise or anticlockwise)

Exercise:

Extend the above solution to print triangulation also. For the above example, the optimal triangulation is 0 3 4, 0 1 3, and 1 2 3.

Sources:

<http://www.cs.utexas.edu/users/djimenez/utsa/cs3343/lecture12.html>

<http://www.cs.utoronto.ca/~heap/Courses/270F02/A4/chains/node2.html>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<http://www.geeksforgeeks.org/minimum-cost-polygon-triangulation/>

Category: [Misc](#) Tags: [Dynamic Programming](#), [geometric algorithms](#)

Post navigation

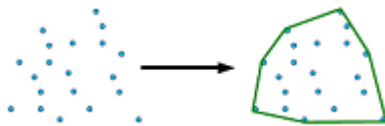
[← D E Shaw Interview | Set 6 \(Off-Campus\) Find n'th number in a number system with only 3 and 4](#) [→](#)

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 8

Convex Hull | Set 1 (Jarvis's Algorithm or Wrapping)

Given a set of points in the plane. the convex hull of the set is the smallest convex polygon that contains all the points of it.



We strongly recommend to see the following post first.

[How to check if two given line segments intersect?](#)

The idea of Jarvis's Algorithm is simple, we start from the leftmost point (or point with minimum x coordinate value) and we keep wrapping points in counterclockwise direction. The big question is, given a point p as current point, how to find the next point in output? The idea is to use [orientation\(\)](#) here. Next point is selected as the point that beats all other points at counterclockwise orientation, i.e., next point is q if for any other point r, we have "orientation(p, r, q) = counterclockwise". Following is the detailed algorithm.

- 1) Initialize p as leftmost point.
- 2) Do following while we don't come back to the first (or leftmost) point.
 -a) The next point q is the point such that the triplet (p, q, r) is counterclockwise for any other point r.
 -b) next[p] = q (Store q as next of p in the output convex hull).
 -c) p = q (Set p as q for next iteration).

```
// A C++ program to find convex hull of a set of points
// Refer http://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/
// for explanation of orientation()
#include <iostream>
using namespace std;

// Define Infinite (Using INT_MAX caused overflow problems)
#define INF 10000

struct Point
{
```

```

    int x;
    int y;
};

// To find orientation of ordered triplet (p, q, r).
// The function returns following values
// 0 --> p, q and r are colinear
// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(Point p, Point q, Point r)
{
    int val = (q.y - p.y) * (r.x - q.x) -
              (q.x - p.x) * (r.y - q.y);

    if (val == 0) return 0; // colinear
    return (val > 0)? 1: 2; // clock or counterclock wise
}

// Prints convex hull of a set of n points.
void convexHull(Point points[], int n)
{
    // There must be at least 3 points
    if (n < 3) return;

    // Initialize Result
    int next[n];
    for (int i = 0; i < n; i++)
        next[i] = -1;

    // Find the leftmost point
    int l = 0;
    for (int i = 1; i < n; i++)
        if (points[i].x < points[l].x)
            l = i;

    // Start from leftmost point, keep moving counterclockwise
    // until reach the start point again
    int p = l, q;
    do
    {
        // Search for a point 'q' such that orientation(p, i, q) is
        // counterclockwise for all points 'i'
        q = (p+1)%n;
        for (int i = 0; i < n; i++)
            if (orientation(points[p], points[i], points[q]) == 2)
                q = i;

        next[p] = q; // Add q to result as a next point of p
        p = q; // Set p as q for next iteration
    } while (p != l);

    // Print Result
    for (int i = 0; i < n; i++)
    {

```

```

        if (next[i] != -1)
            cout << "(" << points[i].x << ", " << points[i].y << ")\n";
    }
}

// Driver program to test above functions
int main()
{
    Point points[] = {{0, 3}, {2, 2}, {1, 1}, {2, 1},
                      {3, 0}, {0, 0}, {3, 3}};
    int n = sizeof(points)/sizeof(points[0]);
    convexHull(points, n);
    return 0;
}

```

Output: The output is points of the convex hull.

```

(0, 3)
(3, 0)
(0, 0)
(3, 3)

```

Time Complexity: For every point on the hull we examine all the other points to determine the next point. Time complexity is $\Theta(m * n)$ where n is number of input points and m is number of output or hull points ($m \geq 2$). The worst case occurs when all the points are on the hull ($m = n$)

We will soon be discussing other algorithms for finding convex hulls.

Sources:

<http://www.cs.uiuc.edu/~jeffe/teaching/373/notes/x05-convexhull.pdf>
<http://www.dcs.gla.ac.uk/~pat/52233/slides/Hull1x1.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

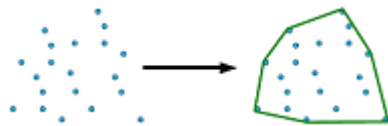
Source

<http://www.geeksforgeeks.org/convex-hull-set-1-jarvis-algorithm-or-wrapping/>

Chapter 9

Convex Hull | Set 2 (Graham Scan)

Given a set of points in the plane. the convex hull of the set is the smallest convex polygon that contains all the points of it.



We strongly recommend to see the following post first.

[How to check if two given line segments intersect?](#)

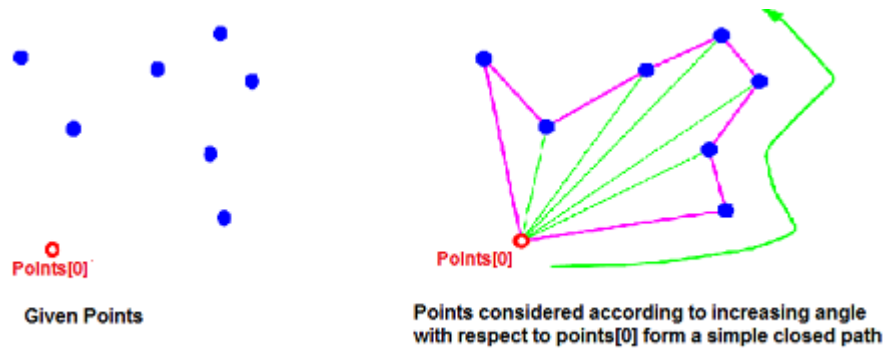
We have discussed [Jarvis's Algorithm](#) for Convex Hull. Worst case time complexity of Jarvis's Algorithm is $O(n^2)$. Using Graham's scan algorithm, we can find Convex Hull in $O(n \log n)$ time. Following is Graham's algorithm

Let `points[0..n-1]` be the input array.

- 1) Find the bottom-most point by comparing y coordinate of all points. If there are two points with same y value, then the point with smaller x coordinate value is considered. Put the bottom-most point at first position.
- 2) Consider the remaining n-1 points and sort them by polar angle in counterclockwise order around `points[0]`. If polar angle of two points is same, then put the nearest point first.
- 3) Create an empty stack 'S' and push `points[0]`, `points[1]` and `points[2]` to S.
- 4) Process remaining n-3 points one by one. Do following for every point '`points[i]`'
 - 4.1) Keep removing points from stack while [orientation](#) of following 3 points is not counterclockwise (or they don't make a left turn).
 - a) Point next to top in stack
 - b) Point at the top of stack
 - c) `points[i]`
 - 4.2) Push `points[i]` to S
- 5) Print contents of S

The above algorithm can be divided in two phases.

Phase 1 (Sort points): We first find the bottom-most point. The idea is to pre-process points by sorting them with respect to the bottom-most point. Once the points are sorted, they form a simple closed path (See following diagram).



What should be the sorting criteria? computation of actual angles would be inefficient since trigonometric functions are not simple to evaluate. The idea is to use the orientation to compare angles without actually computing them (See the `compare()` function below)

Phase 2 (Accept or Reject Points): Once we have the closed path, the next step is to traverse the path and remove concave points on this path. How to decide which point to remove and which to keep? Again, [orientation](#) helps here. The first two points in sorted array are always part of Convex Hull. For remaining points, we keep track of recent three points, and find the angle formed by them. Let the three points be `prev(p)`, `curr(c)` and `next(n)`. If orientation of these points (considering them in same order) is not counterclockwise, we discard `c`, otherwise we keep it. Following diagram shows step by step process of this phase (Source of these diagrams is [Ref 2](#)).

Following is C++ implementation of the above algorithm.

```
// A C++ program to find convex hull of a set of points
// Refer http://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/
// for explanation of orientation()
#include <iostream>
#include <stack>
#include <stdlib.h>
using namespace std;

struct Point
{
    int x;
    int y;
};

// A global point needed for sorting points with reference to the first point
// Used in compare function of qsort()
Point p0;

// A utility function to find next to top in a stack
Point nextToTop(stack<Point> &S)
{
    Point p = S.top();
    S.pop();
    Point res = S.top();
    S.push(p);
    return res;
}
```

```

// A utility function to swap two points
int swap(Point &p1, Point &p2)
{
    Point temp = p1;
    p1 = p2;
    p2 = temp;
}

// A utility function to return square of distance between p1 and p2
int dist(Point p1, Point p2)
{
    return (p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y);
}

// To find orientation of ordered triplet (p, q, r).
// The function returns following values
// 0 --> p, q and r are colinear
// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(Point p, Point q, Point r)
{
    int val = (q.y - p.y) * (r.x - q.x) -
              (q.x - p.x) * (r.y - q.y);

    if (val == 0) return 0; // colinear
    return (val > 0)? 1: 2; // clock or counterclock wise
}

// A function used by library function qsort() to sort an array of
// points with respect to the first point
int compare(const void *vp1, const void *vp2)
{
    Point *p1 = (Point *)vp1;
    Point *p2 = (Point *)vp2;

    // Find orientation
    int o = orientation(p0, *p1, *p2);
    if (o == 0)
        return (dist(p0, *p2) >= dist(p0, *p1))? -1 : 1;

    return (o == 2)? -1: 1;
}

// Prints convex hull of a set of n points.
void convexHull(Point points[], int n)
{
    // Find the bottommost point
    int ymin = points[0].y, min = 0;
    for (int i = 1; i < n; i++)
    {
        int y = points[i].y;

        // Pick the bottom-most or chose the left most point in case of tie

```

```

        if ((y < ymin) || (ymin == y && points[i].x < points[min].x))
            ymin = points[i].y, min = i;
    }

    // Place the bottom-most point at first position
    swap(points[0], points[min]);

    // Sort n-1 points with respect to the first point. A point p1 comes
    // before p2 in sorted output if p2 has larger polar angle (in
    // counterclockwise direction) than p1
    p0 = points[0];
    qsort(&points[1], n-1, sizeof(Point), compare);

    // Create an empty stack and push first three points to it.
    stack<Point> S;
    S.push(points[0]);
    S.push(points[1]);
    S.push(points[2]);

    // Process remaining n-3 points
    for (int i = 3; i < n; i++)
    {
        // Keep removing top while the angle formed by points next-to-top,
        // top, and points[i] makes a non-left turn
        while (orientation(nextToTop(S), S.top(), points[i]) != 2)
            S.pop();
        S.push(points[i]);
    }

    // Now stack has the output points, print contents of stack
    while (!S.empty())
    {
        Point p = S.top();
        cout << "(" << p.x << ", " << p.y << ")" << endl;
        S.pop();
    }
}

// Driver program to test above functions
int main()
{
    Point points[] = {{0, 3}, {1, 1}, {2, 2}, {4, 4},
                      {0, 0}, {1, 2}, {3, 1}, {3, 3}};
    int n = sizeof(points)/sizeof(points[0]);
    convexHull(points, n);
    return 0;
}

```

Output:

```

(0, 3)
(4, 4)
(3, 1)
(0, 0)

```

Time Complexity: Let n be the number of input points. The algorithm takes $O(n \log n)$ time if we use a $O(n \log n)$ sorting algorithm.

The first step (finding the bottom-most point) takes $O(n)$ time. The second step (sorting points) takes $O(n \log n)$ time. In third step, every element is pushed and popped at most one time. So the third step to process points one by one takes $O(n)$ time, assuming that the stack operations take $O(1)$ time. Overall complexity is $O(n) + O(n \log n) + O(n)$ which is $O(n \log n)$

References:

[Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](#)

<http://www.dcs.gla.ac.uk/~pat/52233/slides/Hull1x1.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<http://www.geeksforgeeks.org/convex-hull-set-2-graham-scan/>

Category: [Misc](#) Tags: [geometric algorithms](#), [MathematicalAlgo](#)

Post navigation

[← Stable Marriage Problem Amazon Interview | Set 34](#) [→](#)

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.