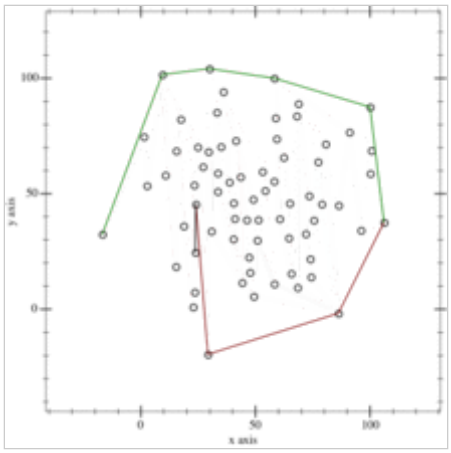
Algorithm Implementation/Geometry/Convex hull/Monotone chain

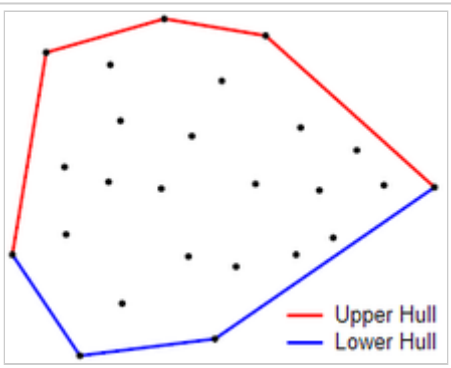
Andrew's monotone chain convex hull algorithm constructs the convex hull of a set of 2-dimensional points in $O(n \log n)$ time.

It does so by first sorting the points lexicographically (first by *x*-coordinate, and in case of a tie, by *y*-coordinate), and then constructing upper and lower hulls of the points in $O(n)$ time.

An upper hull is the part of the convex hull, which is visible from the above. It runs from its rightmost point to the leftmost point in counterclockwise order. Lower hull is the remaining part of the convex hull.



Animation depicting the Monotone convex hull algorithm



Upper and lowers hulls of a set of points

Contents

Pseudo-code

```
Input: a list P of points in the plane.

Sort the points of P by x-coordinate (in case of a tie, sort by y-coordinate).

Initialize U and L as empty lists.
The lists will hold the vertices of upper and lower hulls respectively.

for i = 1, 2, ..., n:
    while L contains at least two points and the sequence of last two points
             of L and the point P[i] does not make a counter-clockwise turn:
        remove the last point from L
    append P[i] to L

for i = n, n-1, ..., 1:
    while U contains at least two points and the sequence of last two points
             of U and the point P[i] does not make a counter-clockwise turn:
        remove the last point from U
```

```
    append P[i] to U

Remove the last point of each list (it's the same as the first point of the other list).
Concatenate L and U to obtain the convex hull of P.
Points in the result will be listed in counter-clockwise order.
```

JavaScript

```
function cross(o, a, b) {
   return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])
}

/**
 * @param points An array of [X, Y] coordinates
 */
function convexHull(points) {
   points.sort(function(a, b) {
      return a[0] == b[0] ? a[1] - b[1] : a[0] - b[0];
   });

   var lower = [];
   for (var i = 0; i < points.length; i++) {
      while (lower.length >= 2 && cross(lower[lower.length - 2], lower[lower.length - 1], points[i]) <= 0) {
         lower.pop();
      }
      lower.push(points[i]);
   }

   var upper = [];
   for (var i = points.length - 1; i >= 0; i--) {
      while (upper.length >= 2 && cross(upper[upper.length - 2], upper[upper.length - 1], points[i]) <= 0) {
         upper.pop();
      }
      upper.push(points[i]);
   }

   upper.pop();
   lower.pop();
   return lower.concat(upper);
}
```

Java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Arrays;
import java.util.StringTokenizer;

class Point implements Comparable<Point> {
    int x, y;

    public int compareTo(Point p) {
        if (this.x == p.x) {
            return this.y - p.y;
        } else {
            return this.x - p.x;
        }
    }

    public String toString() {
        return "("+x + "," + y+")";
    }

}

public class ConvexHull {
```

```
    public static long cross(Point O, Point A, Point B) {
        return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
    }

    public static Point[] convex_hull(Point[] P) {

        if (P.length > 1) {
            int n = P.length, k = 0;
            Point[] H = new Point[2 * n];

            Arrays.sort(P);

            // Build lower hull
            for (int i = 0; i < n; ++i) {
                while (k >= 2 && cross(H[k - 2], H[k - 1], P[i]) <= 0)
                    k--;
                H[k++] = P[i];
            }

            // Build upper hull
            for (int i = n - 2, t = k + 1; i >= 0; i--) {
                while (k >= t && cross(H[k - 2], H[k - 1], P[i]) <= 0)
                    k--;
                H[k++] = P[i];
            }
            if (k > 1) {
                H = Arrays.copyOfRange(H, 0, k - 1); // remove non-hull vertices after k; remove k - 1 which is a duplicate
            }
            return H;
        } else if (P.length <= 1) {
            return P;
        } else{
            return null;
        }
    }

    public static void main(String[] args) throws IOException {

        BufferedReader f = new BufferedReader(new FileReader("hull.in"));   // "hull.in"  Input Sample => size x y x y x y x y
        StringTokenizer st = new StringTokenizer(f.readLine());
        Point[] p = new Point[Integer.parseInt(st.nextToken())];
        for (int i = 0; i < p.length; i++) {
            p[i] = new Point();
            p[i].x = Integer.parseInt(st.nextToken()); // Read X coordinate
            p[i].y = Integer.parseInt(st.nextToken()); // Read y coordinate
        }

        Point[] hull = convex_hull(p).clone();

        for (int i = 0; i < hull.length; i++) {
            if (hull[i] != null)
                System.out.print(hull[i]);
        }
    }

}
```

Python

```
def convex_hull(points):
    """Computes the convex hull of a set of 2D points.

    Input: an iterable sequence of (x, y) pairs representing the points.
    Output: a list of vertices of the convex hull in counter-clockwise order,
      starting from the vertex with the lexicographically smallest coordinates.
    Implements Andrew's monotone chain algorithm. O(n log n) complexity.
    """
```

```
    # Sort the points lexicographically (tuples are compared lexicographically).
    # Remove duplicates to detect the case we have just one unique point.
    points = sorted(set(points))

    # Boring case: no points or a single point, possibly repeated multiple times.
    if len(points) <= 1:
        return points

    # 2D cross product of OA and OB vectors, i.e. z-component of their 3D cross product.
    # Returns a positive value, if OAB makes a counter-clockwise turn,
    # negative for clockwise turn, and zero if the points are collinear.
    def cross(o, a, b):
        return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])

    # Build lower hull
    lower = []
    for p in points:
        while len(lower) >= 2 and cross(lower[-2], lower[-1], p) <= 0:
            lower.pop()
        lower.append(p)

    # Build upper hull
    upper = []
    for p in reversed(points):
        while len(upper) >= 2 and cross(upper[-2], upper[-1], p) <= 0:
            upper.pop()
        upper.append(p)

    # Concatenation of the lower and upper hulls gives the convex hull.
    # Last point of each list is omitted because it is repeated at the beginning of the other list.
    return lower[:-1] + upper[:-1]


# Example: convex hull of a 10-by-10 grid.
assert convex_hull([(i//10, i%10) for i in range(100)]) == [(0, 0), (9, 0), (9, 9), (0, 9)]
```

Ruby

```
# the python code converted to ruby syntax
def convex_hull(points)
  points.sort!.uniq!
  return points if points.length <= 3
  def cross(o, a, b)
    (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])
  end
  lower = Array.new
  points.each{|p|
    while lower.length > 1 and cross(lower[-2], lower[-1], p) <= 0 do lower.pop end
    lower.push(p)
  }
  upper = Array.new
  points.reverse_each{|p|
    while upper.length > 1 and cross(upper[-2], upper[-1], p) <= 0 do upper.pop end
    upper.push(p)
  }
  return lower[0...-1] + upper[0...-1]
end
fail unless convex_hull((0..9).to_a.repeated_permutation(2).to_a) == [[0, 0], [9, 0], [9, 9], [0, 9]]
```

C++

```
// Implementation of Andrew's monotone chain 2D convex hull algorithm.
// Asymptotic complexity: O(n log n).
// Practical performance: 0.5-1.0 seconds for n=1000000 on a 1GHz machine.
#include <algorithm>
#include <vector>
using namespace std;
```

```
typedef double coord_t;         // coordinate type
typedef double coord2_t;  // must be big enough to hold 2*max(|coordinate|)^2

struct Point {
    coord_t x, y;

    bool operator <(const Point &p) const {
        return x < p.x || (x == p.x && y < p.y);
    }
};

// 2D cross product of OA and OB vectors, i.e. z-component of their 3D cross product.
// Returns a positive value, if OAB makes a counter-clockwise turn,
// negative for clockwise turn, and zero if the points are collinear.
coord2_t cross(const Point &O, const Point &A, const Point &B)
{
    return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}

// Returns a list of points on the convex hull in counter-clockwise order.
// Note: the last point in the returned list is the same as the first one.
vector<Point> convex_hull(vector<Point> P)
{
    int n = P.size(), k = 0;
    vector<Point> H(2*n);

    // Sort points lexicographically
    sort(P.begin(), P.end());

    // Build lower hull
    for (int i = 0; i < n; ++i) {
        while (k >= 2 && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
        H[k++] = P[i];
    }

    // Build upper hull
    for (int i = n-2, t = k+1; i >= 0; i--) {
        while (k >= t && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
        H[k++] = P[i];
    }

    H.resize(k-1);
    return H;
}
```

Perl

```
use Sort::Key::Radix qw(nkeysort); # use radix sort for an O(n) algorithm

sub convex_hull {
    return @_ if @_ < 2;

    my @p = nkeysort { $_->[0] } @_;

    my (@u, @l);
    my $i = 0;
    while ($i < @p) {
        my $iu = my $il = $i;
        my ($x, $yu) = @{$p[$i]};
        my $yl = $yu;
        # search for upper and lower Y for the current X
        while (++$i < @p and $p[$i][0] == $x) {
            my $y = $p[$i][1];
            if ($y < $yl) {
                $il = $i;
                $yl = $y;
            }
            elsif ($y > $yu) {
                $iu = $i;
                $yu = $y;
```

```
            }
        }
        while (@l >= 2) {
            my ($ox, $oy) = @{$l[-2]};
            last if ($l[-1][1] - $oy) * ($x - $ox) < ($yl - $oy) * ($l[-1][0] - $ox);
            pop @l;
        }
        push @l, $p[$il];
        while (@u >= 2) {
            my ($ox, $oy) = @{$u[-2]};
            last if ($u[-1][1] - $oy) * ($x - $ox) > ($yu - $oy) * ($u[-1][0] - $ox);
            pop @u;
        }
        push @u, $p[$iu];
    }

    # remove points from the upper hull extremes when they are already
    # on the lower hull:
    shift @u if $u[0][1] == $l[0][1];
    pop @u if @u and $u[-1][1] == $l[-1][1];

    return (@l, reverse @u);
}
```

C

C sources taken from the Math::ConvexHull::MonotoneChain Perl module. Note that this implementation works on sorted input points. Otherwise, it's rather similar to the C++ implementation above.

```
typedef struct {
  double x;
  double y;
} point_t;

typedef point_t* point_ptr_t;

/* Three points are a counter-clockwise turn if ccw > 0, clockwise if
 * ccw < 0, and collinear if ccw = 0 because ccw is a determinant that
 * gives the signed area of the triangle formed by p1, p2 and p3.
 */
static double
ccw(point_t* p1, point_t* p2, point_t* p3)
{
  return (p2->x - p1->x)*(p3->y - p1->y) - (p2->y - p1->y)*(p3->x - p1->x);
}

/* Returns a list of points on the convex hull in counter-clockwise order.
 * Note: the last point in the returned list is the same as the first one.
 */
void
convex_hull(point_t* points, ssize_t npoints, point_ptr_t** out_hull, ssize_t* out_hullsize)
{
  point_ptr_t* hull;
  ssize_t i, t, k = 0;

  hull = *out_hull;

  /* lower hull */
  for (i = 0; i < npoints; ++i) {
    while (k >= 2 && ccw(hull[k-2], hull[k-1], &points[i]) <= 0) --k;
    hull[k++] = &points[i];
  }

  /* upper hull */
  for (i = npoints-2, t = k+1; i >= 0; --i) {
    while (k >= t && ccw(hull[k-2], hull[k-1], &points[i]) <= 0) --k;
    hull[k++] = &points[i];
  }

  *out_hull = hull;
```

```
  *out_hullsize = k;
}
```

Haskell

```haskell
import Data.List (sort)

-- Coordinate type
type R = Double

-- Vector / point type
type R2 = (R, R)

-- Checks if it's shortest to rotate from the OA to the OB vector in a clockwise
-- direction.
clockwise :: R2 -> R2 -> R2 -> Bool
clockwise o a b = (a `sub` o) `cross` (b `sub` o) <= 0

-- 2D cross product.
cross :: R2 -> R2 -> R
cross (x1, y1) (x2, y2) = x1 * y2 - x2 * y1

-- Subtract two vectors.
sub :: R2 -> R2 -> R2
sub (x1, y1) (x2, y2) = (x1 - x2, y1 - y2)

-- Implements the monotone chain algorithm
convexHull :: [R2] -> [R2]
convexHull [] = []
convexHull [p] = [p]
convexHull points = lower ++ upper
  where
    sorted = sort points
    lower = chain sorted
    upper = chain (reverse sorted)

chain :: [R2] -> [R2]
chain = go []
  where
    -- The first parameter accumulates a monotone chain where the most recently
    -- added element is at the front of the list.
    go :: [R2] -> [R2] -> [R2]
    go acc@(r1:r2:rs) (x:xs) =
      if clockwise r2 r1 x
        -- Made a clockwise turn - remove the most recent part of the chain.
        then go (r2:rs) (x:xs)
        -- Made a counter-clockwise turn - append to the chain.
        else go (x:acc) xs
    -- If there's only one point in the chain, just add the next visited point.
    go acc (x:xs) = go (x:acc) xs
    -- No more points to consume - finished!  Note: the reverse here causes the
    -- result to be consistent with the other examples (a ccw hull), but
    -- removing that and using (upper ++ lower) above will make it cw.
    go acc [] = reverse $ tail acc

main :: IO ()
main =
    if result == expected
      then putStrLn "convexHull worked!"
      else fail $ "convexHull broken, got " ++ show result
  where
    result = convexHull [(x, y) | x <- [0..9], y <- [0..9]]
    expected = [(0, 0), (9, 0), (9, 9), (0, 9)]
```

Elixir

```
defmodule ConvexHull do
  @type coordinate :: {number, number}
  @type point_list :: [coordinate]

  @spec find(point_list) :: point_list
  def find(points) do
    sorted = points |> Enum.sort
    left = sorted |> do_half_calc
    right = sorted |> Enum.reverse |> do_half_calc
    [left, right] |> Enum.concat
  end

  @spec do_half_calc(point_list) :: point_list
  defp do_half_calc(points) do
    points
    |> Enum.reduce([], &add_to_convex_list/2)
    |> tl
    |> Enum.reverse
  end

  @spec perp_prod(coordinate, coordinate, coordinate) :: number
  defp perp_prod({x0, y0}, {x1, y1}, {x2, y2}) do
    (x1 - x0) * (y2 - y0) - (y1 - y0) * (x2 - x0)
  end

  @spec add_to_convex_list(coordinate, point_list) :: point_list
  defp add_to_convex_list(p2, list) do
    {:ok, new_tail} =
      list
      |> stream_tails
      |> Stream.drop_while(fn tail -> oa_left_of_ob(p2, tail) end)
      |> Enum.fetch(0)
    [p2 | new_tail]
  end

  @spec oa_left_of_ob(coordinate, point_list) :: number
  defp oa_left_of_ob(b, [a, o | _]), do: perp_prod(o, a, b) <= 0
  defp oa_left_of_ob(_, _), do: false

  @spec stream_tails(point_list) :: Enum.t(point_list)
  defp stream_tails(list) do
    tails = Stream.unfold(list, fn [_ | t] -> {t, t} end)
    Stream.concat([list], tails)
  end

end


ExUnit.start

defmodule ConvexHullStart do
  use ExUnit.Case, async: true

  @points for x <- 0..9, y <- 0..9, do: {x, y}

  test "finding the convex hull for all points (0,0) .. (9,9)" do
    assert ConvexHull.find(@points) == [{0, 0}, {9, 0}, {9, 9}, {0, 9}]
  end
end
```

Clojure

```
(defn convex-hull
"The function convex-hull accepts a collection of vectors containing pairs of numbers
 which define the [x y] bounds of the polygon,
 e.g. [[0 0] [0 1] [0 2] [0 3]
       [1 0] [1 1] [1 2] [1 3]
       [2 0] [2 1] [2 2] [2 3] [2 4] [2 5] [2 6] [2 7] [2 8] [2 9]
       [3 0] [3 1] [3 2] [3 3]
       [4 0] [4 1] [4 2] [4 3] [4 4]
```

```
            [5 1]]

When run on the above collection of points the output will be
  [[0 0] [0 3] [2 9] [4 4] [5 1] [4 0]]

The input collection of vectors can be either a vector ( e.g. [[0 0] [1 1] [2 2]]) or a quoted sequence
(e.g. '([0 0] [1 1] [2 2]) or (quote ([0 0] [1 1] [2 2])).

The output collection will always be a vector of vectors (e.g. [[0 0] [0 9] [9 9] [9 0]]) which
define the bounds of the convex hull."
 [c]
 (let [cc         (sort-by identity c)
       x          #(first %)    ; returns the 'x' coordinate from an [x y] vector
       y          #(second %)   ; returns the 'y' coordinate from an [x y] vector
       ccw        #(- (* (- (x %2) (x %1)) (- (y %3) (y %1))) (* (- (y %2) (y %1)) (- (x %3) (x %1))))
       half-hull  #(loop [h  [] ; returns the upper half-hull of the collection of vectors
                          c  %]
                     (if (not (empty? c))
                       (if (and (> (count h) 1) (<= 0 (ccw (h (- (count h) 2)) (h (- (count h) 1)) (first c))))
                         (recur (vec (butlast h)) c)
                         (recur (conj h (first c)) (rest c)))
                       h))
      upper-hull  (butlast (half-hull cc))
      lower-hull  (butlast (half-hull (reverse cc)))]
   (vec (concat upper-hull lower-hull ))))
```

Scala

```
import scala.collection.mutable.ArrayBuffer

 def convexHull(points: ArrayBuffer[(Int, Int)]): ArrayBuffer[(Int, Int)] = {


   // 2D cross product of OA and OB vectors, i.e. z-component of their 3D cross product.
   // Returns a positive value, if OAB makes a counter-clockwise turn,
   // negative for clockwise turn, and zero if the points are collinear.
   def cross(o: (Int, Int), a: (Int, Int), b: (Int, Int)): Int = {
     (a._1 - o._1) * (b._2 - o._2) - (a._2 - o._2) * (b._1 - o._1)
   }

   val distinctPoints = points.distinct

   // No sorting needed if there are less than 2 unique points.
   if(distinctPoints.length < 2) {
     return points
   } else {

     val sortedPoints = distinctPoints.sorted

     // Build the lower hull
     val lower = ArrayBuffer[(Int, Int)]()
     for(i <- sortedPoints){
       while(lower.length >= 2 && cross(lower(lower.length - 2), lower(lower.length - 1) , i) <= 0){
         lower -= lower.last
       }
       lower += i
     }

     // Build the upper hull
     val upper = ArrayBuffer[(Int, Int)]()
     for(i <- sortedPoints.reverse){
       while(upper.size >= 2 && cross(upper(upper.length - 2), upper(upper.length - 1) , i) <= 0){
         upper -= upper.last
       }
       upper += i
     }

     // Last point of each list is omitted because it is repeated at the beginning of the other list.
     lower -= lower.last
     upper -= upper.last
```

```
    // Concatenation of the lower and upper hulls gives the convex hull
    return lower ++= upper
  }
}
```

References

- De Berg, van Kreveld, Overmars, Schwarzkopf. Computational Geometry: Algorithms and Applications. 2nd edition, Springer-Verlag. ISBN 3540656200.
- Dan Sunday. The Convex Hull of a 2D Point Set or Polygon (http://softsurfer.com/Archive/algorithm_0109/algorithm_0109.htm). (archived copy (http://www.webcitation.org/5uY0uFNqR) by webcite)
- A. M. Andrew, "Another Efficient Algorithm for Convex Hulls in Two Dimensions", Info. Proc. Letters 9, 216-219 (1979).