```
from which_pyqt import PYQT_VER

if PYQT_VER == 'PYQT5':
    from PyQt5.QtCore import QLineF, QPointF, QObject
elif PYQT_VER == 'PYQT4':
    from PyQt4.QtCore import QLineF, QPointF, QObject
elif PYQT_VER == 'PYQT6':
    from PyQt6.QtCore import QLineF, QPointF, QObject
else:
    raise Exception('Unsupported Version of PyQt: {}'.format(PYQT_VER))

import time
import math

# Some global color constants that might be useful
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)

# Global variable that controls the speed of the recursion automation, in seconds
#
PAUSE = 0.1

#
# This is the class you have to complete.
#
class ConvexHullSolver(QObject):

    # Class constructor
    def __init__(self):
        super().__init__()
        self.pause = False

    # Some helper methods that make calls to the GUI, allowing us to send updates
    # to be displayed.

    def showTangent(self, line, color):
        lines = [line]
        self.view.addLines(lines, color)
        if self.pause:
            time.sleep(PAUSE)

    def eraseTangent(self, line):
        self.view.clearLines([line])

    def blinkTangent(self, line, color):
        self.showTangent(line, color)
        self.eraseTangent(line)

    def showHull(self, polygon, color):
        self.view.addLines(polygon, color)
        if self.pause:
            time.sleep(PAUSE)

    def eraseHull(self, polygon):
        self.view.clearLines(polygon)

    def showText(self, text):
        self.view.displayStatusText(text)

    # This is the method that gets called by the GUI and actually executes
    # the finding of the hull
    def compute_hull(self, points, pause, view):
        self.pause = pause
        self.view = view
        assert (type(points) == list and type(points[0]) == QPointF)

        t1 = time.time()
        points.sort(key=lambda p: p.x())  # sort function for list in python is O(n log n)
        t2 = time.time()

        t3 = time.time()

        hull_points = self.create_hull(points)
        t4 = time.time()

        # when passing lines to the display, pass a list of QLineF objects.  Each QLineF
        # object can be created with two QPointF objects corresponding to the endpoints
        polygon = [QLineF(hull_points[i], hull_points[(i + 1) % len(hull_points)]) for i in range(len(hull_points))]
        self.showHull(polygon, RED)
        self.showText('Time Elapsed (Convex Hull): {:3.3f} sec'.format(t4 - t1))

    # Time complexity: Divide and conquer, so use master theorem. Dividing into two subproblems (a = 2),
    #   each has half the size (b = 2), merging them back together in linear time (d = 1). 2 / (2^1) = 1, giving us
    #   O(n log n) time
    # Space complexity: O(n). The recursion tree is iterating down one branch of the tree and you aren't creating any
    #   new lists at each node. The max you could ever have at once is n memory used.
    def create_hull(self, points):
        # recurse down, splitting the points in half, till you have 2 or 3 point elements
        if len(points) > 3:
            middle = math.floor(len(points) / 2)
            l_hull = self.create_hull(points[:middle])
            r_hull = self.create_hull(points[middle:])
        elif len(points) == 3:
            # if it's 3 points, sort them in clockwise order
            return self.sort_base_case_clockwise(points)
        else:
            # else it is 2 points, and there is no need to sort because we already sorted L to R
            return points

        # polygon_l = [QLineF(l_hull[i], l_hull[(i + 1) % len(l_hull)]) for i in range(len(l_hull))]
        # self.showHull(polygon_l, RED)
        # polygon_r = [QLineF(r_hull[i], r_hull[(i + 1) % len(r_hull)]) for i in range(len(r_hull))]
        # self.showHull(polygon_r, RED)

        # find the R most index in l_hull and the L most index in r_hull (which is just the first one)
        current_index_l = self.find_right_most_index(l_hull)
        current_index_r = 0

        # call the helper functions to find the upper bounds and lower bounds, then save them as their point indexes
        up_left_index, up_right_index = self.find_upper_bound(l_hull, r_hull, current_index_l, current_index_r)
        down_left_index, down_right_index = self.find_lower_bound(l_hull, r_hull, current_index_l, current_index_r)

        # create the hull that will be returned. this is done by circling clockwise around the edges of the hull,
        # dropping the points that are in the middle using the indexes
        # 0 --> up_left_index --> up_right_index --> down_right_index --> down_left_index --> back to 0
        hull_points = []
        hull_points.extend(l_hull[:up_left_index + 1])
        if down_right_index != 0:
            hull_points.extend(r_hull[up_right_index:down_right_index + 1])
        else:
            hull_points.extend(r_hull[up_right_index:])
            hull_points.append(r_hull[0])
        if down_left_index != 0:
            hull_points.extend(l_hull[down_left_index:])

        # self.eraseHull(polygon_l)
        # self.eraseHull(polygon_r)
        # polygon = [QLineF(hull_points[i], hull_points[(i + 1) % len(hull_points)]) for i in range(len(hull_points))]
        # self.showHull(polygon, GREEN)

        return hull_points

    @staticmethod
    def slope(x1, y1, x2, y2):
        # returns the slope given the x / y values of two points
        return (y2 - y1) / (x2 - x1)

    def sort_base_case_clockwise(self, points):
        # 3 points need to be sorted in clockwise order
        sorted_points = [points[0]]
        slope_1 = self.slope(QPointF.x(points[0]),
                             QPointF.y(points[0]),
                             QPointF.x(points[1]),
                             QPointF.y(points[1]))
        slope_2 = self.slope(QPointF.x(points[0]),
                             QPointF.y(points[0]),
                             QPointF.x(points[2]),
                             QPointF.y(points[2]))
        # compare the slopes between the first and second/first and third
        if slope_1 > slope_2:
            sorted_points.extend([points[1], points[2]])
        else:
            sorted_points.extend([points[2], points[1]])

        return sorted_points

    # Time complexity: worst case scenario would be iterating through the whole hull, so O(n)
    # Space complexity: only storing the list of points, so O(n)
    @staticmethod
    def find_right_most_index(l_hull):
        right_most_index = 0
        # iterate through the points in the l_hull--if the pt's x is bigger, keep going. else return the index.
        for index, pt in enumerate(l_hull):
            if index == 0:
                continue

            if QPointF.x(pt) > QPointF.x(l_hull[right_most_index]):
                right_most_index = index
            else:
                return right_most_index

        return right_most_index

    # Time complexity: the nested while loops are the part that will take the longest; however, the inner while loops
    #   will maximally terminate after looping through all the points (the slope must fail the check if it loops back
    #   to the start of the hull). So the inner loops are both O(n), and the outer loop terminates if neither inner
    #   loop changes, meaning that this is O(n + n) --> O(n).
    # Space complexity: the only thing stored is both lists of hulls, so O(n + n) --> O(n).
    def find_upper_bound(self, l_hull, r_hull, current_index_l, current_index_r):
        upper_bound_slope = self.slope(QPointF.x(l_hull[current_index_l]),
                                       QPointF.y(l_hull[current_index_l]),
                                       QPointF.x(r_hull[current_index_r]),
                                       QPointF.y(r_hull[current_index_r]))

        # line = QLineF(l_hull[current_index_l], r_hull[current_index_r])
        # self.blinkTangent(line, BLUE)

        # nested while loop--outer loop continues if there has been a change in the upper_bound_slope, inner loops move
        # the indexes of the l_hull and r_hull around to find the best slope, respectively.
        change = True
        while change:
            change = False

            while True:
                check_index_l = (current_index_l - 1) % len(l_hull)
                check_slope = self.slope(QPointF.x(l_hull[check_index_l]),
                                         QPointF.y(l_hull[check_index_l]),
                                         QPointF.x(r_hull[current_index_r]),
                                         QPointF.y(r_hull[current_index_r]))

                # line = QLineF(l_hull[check_index_l], r_hull[current_index_r])
                # self.blinkTangent(line, BLUE)

                if check_slope > upper_bound_slope:
                    # end the loop--we've found the best slope for the current_index_r
                    break
                else:
                    # keep going--save the check variables into the current variables and try again
                    change = True
                    upper_bound_slope = check_slope
                    current_index_l = check_index_l

            while True:
                check_index_r = (current_index_r + 1) % len(r_hull)
                check_slope = self.slope(QPointF.x(l_hull[current_index_l]),
                                         QPointF.y(l_hull[current_index_l]),
                                         QPointF.x(r_hull[check_index_r]),
                                         QPointF.y(r_hull[check_index_r]))

                # line = QLineF(l_hull[current_index_l], r_hull[check_index_r])
                # self.blinkTangent(line, BLUE)

                if check_slope < upper_bound_slope:
                    # end the loop--we've found the best slope for the current_index_l
                    break
                else:
                    # keep going--save the check variables into the current variables and try again
                    change = True
                    upper_bound_slope = check_slope
                    current_index_r = check_index_r

        # line = QLineF(l_hull[current_index_l], r_hull[current_index_r])
        # self.blinkTangent(line, GREEN)
        # self.blinkTangent(line, GREEN)
        return current_index_l, current_index_r

    # Time complexity: the nested while loops are the part that will take the longest; however, the inner while loops
    #   will maximally terminate after looping through all the points (the slope must fail the check if it loops back
    #   to the start of the hull). So the inner loops are both O(n), and the outer loop terminates if neither inner
    #   loop changes, meaning that this is O(n + n) --> O(n).
    # Space complexity: the only thing stored is both lists of hulls, so O(n + n) --> O(n).
    def find_lower_bound(self, l_hull, r_hull, current_index_l, current_index_r):
        lower_bound_slope = self.slope(QPointF.x(l_hull[current_index_l]),
                                       QPointF.y(l_hull[current_index_l]),
                                       QPointF.x(r_hull[current_index_r]),
                                       QPointF.y(r_hull[current_index_r]))

        # line = QLineF(l_hull[current_index_l], r_hull[current_index_r])
        # self.blinkTangent(line, BLUE)

        # double while loop--outer loop continues if there has been a change in the upper_bound_slope, inner loops move
        # the indexes of the l_hull and r_hull around to find the best slope, respectively.
        change = True
        while change:
            change = False

            while True:
                check_index_l = (current_index_l + 1) % len(l_hull)
                check_slope = self.slope(QPointF.x(l_hull[check_index_l]),
                                         QPointF.y(l_hull[check_index_l]),
                                         QPointF.x(r_hull[current_index_r]),
                                         QPointF.y(r_hull[current_index_r]))

                # line = QLineF(l_hull[check_index_l], r_hull[current_index_r])
                # self.blinkTangent(line, BLUE)

                if check_slope < lower_bound_slope:
                    # end the loop--we've found the best slope for the current_index_r
                    break
                else:
                    # keep going--save the check variables into the current variables and try again
                    change = True
                    lower_bound_slope = check_slope
                    current_index_l = check_index_l

            while True:
                check_index_r = (current_index_r - 1) % len(r_hull)
                check_slope = self.slope(QPointF.x(l_hull[current_index_l]),
                                         QPointF.y(l_hull[current_index_l]),
                                         QPointF.x(r_hull[check_index_r]),
                                         QPointF.y(r_hull[check_index_r]))

                # line = QLineF(l_hull[current_index_l], r_hull[check_index_r])
                # self.blinkTangent(line, BLUE)

                if check_slope > lower_bound_slope:
                    # end the loop--we've found the best slope for the current_index_l
                    break
                else:
                    # keep going--save the check variables into the current variables and try again
                    change = True
                    lower_bound_slope = check_slope
                    current_index_r = check_index_r

        # line = QLineF(l_hull[current_index_l], r_hull[current_index_r])
        # self.blinkTangent(line, GREEN)
        # self.blinkTangent(line, GREEN)
        return current_index_l, current_index_r
```