Question 1 (Look for the big O above each function)

```
# Uncomment this line to import some functions that can help
# you debug your algorithm
import matplotlib.pyplot as plt
from plotting import draw_line, draw_hull, circle_point
plt.ion()
```

O(n), Grows based on the length of points

```
def find_average_x(points: list[tuple[float, float]]) -> float:
    average_x = 0.0
    for tup in points:
        average_x += tup[0]
    average_x /= len(points)
    return average_x
```

O(n), Grows based on the length of points

```
def split_into_left_right(points: list[tuple[float, float]], left: list[tuple[float, float]],
                          right: list[tuple[float, float]],
                          avg_x: float) -> tuple[list[tuple[float, float]], list[tuple[float, float]]]:

    for tpl in points:
        if tpl[0] < avg_x:
            left.append(tpl)
        else:
            right.append(tpl)
```

```
    return (left, right)
```

O(log(n)), The number of points that we have to look at is always less than the size of Points, but it is not quite constant

```
def find_extreme_point(points: list[tuple[float, float]], find_right_most: bool) -> tuple[float, float]:


    if find_right_most:
        return max(points, key=lambda p: p[0])
    else:
        return min(points, key=lambda p: p[0])
```

O(1), just a quick calculation that is not based on the size of any list

```
def calculate_slope(left_point: tuple[float,float], right_point: tuple[float,float]) -> float:
    return (right_point[1] - left_point[1])/ (right_point[0] - left_point[0])
```

O(1), just a quick comparison that is not based on the size of any list

```
def is_bigger_tangent_line(temp: float, curr_slope: float) -> bool:
    return temp < curr_slope
```

O(1), just a quick calculation that is not based on the size of any list

```
def is_smaller_tangent_line(temp: float, curr_slope: float) -> bool:
    return temp > curr_slope
```

O(log(n)), I think the part that is slowest us down the most is finding the index of the the starting point. The modulus calculation doesn't require a lot of computation power

```
def find_neighbor(points: list[tuple[float, float]], current_point: tuple[float, float],
                  isClockwise: bool) -> tuple[float, float]:
```

'''This will find the neighor, allowing for either clockwise or counter clockwise'''

O(1)/O(log(n)) -> It could go either way, but I would say O(log(n)) is probably closer to the actual average case.

idx = points.index(current_point)

if isClockwise:

O(1) -> len(points) is the only thing that makes me hesitate, but I will say that counting the points is of constant time. So the only thing left to consider is the modulus calculation, which is constant.

return points[(idx - 1) % len(points)]

else:

O(1) -> len(points) is the only thing that makes me hesitate, but I will say that counting the points is of constant time. So the only thing left to consider is the modulus calculation, which is constant.

return points[(idx + 1) % len(points)]

4 * O(log(n) = O(log(n)), there are 4 different pieces that all take log(n), and everything is of less complexity

```
def find_upper_Tangent(left: list[tuple[float, float]],
                       right: list[tuple[float, float]]) -> tuple[tuple[float, float], tuple[float, float]]:
```

O(log(n)) -> see explanation from above

p: tuple[float, float] = find_extreme_point(left, True)

O(log(n)) -> see explanation from above

q: tuple[float, float] = find_extreme_point(right, False)

O(n) -> see explanation from above

temp: float = calculate_slope(p, q)

O(1)

done: bool = False

O(1)

if len(left) == 1 and len(right) == 1:

return (p, q)

while not done:

done = True

O(log(n) -> the thing that is slowing us down the most is finding_neighbor

while True:

O(log(n) -> see explanation from above

r = find_neighbor(left, p, isClockwise=False)

O(1)

if r == p:

break

O(1)

prev_slope = temp

O(1)

curr_slope = calculate_slope(r, q)

O(1): See above explanation

if is_smaller_tangent_line(prev_slope, curr_slope):

O(1)

temp = curr_slope

O(1)

p = r

O(1)

```
            done = False
        else:
            break
```

O(log(n) -> the steps are nearly identical to the previous “while loop”, we just loop in the opposite direction

```
    while True:
        v = find_neighbor(right, q, isClockwise=True)
        if v == q:
            break
        prev_slope = temp
        curr_slope = calculate_slope(v, p)

        if is_bigger_tangent_line(prev_slope, curr_slope):
            temp = curr_slope
            q = v
            done = False
        else:
            break

  return (p, q)
```

O(log(n), see find_Upper_Tanget

```
def find_Lower_Tangent(left: list[tuple[float, float]],
                       right: list[tuple[float, float]]) -> tuple[tuple[float, float], tuple[float, float]]:
```

```
p: tuple[float, float] = find_extreme_point(left, True)
q: tuple[float, float] = find_extreme_point(right, False)
temp: float = calculate_slope(p, q)
done: bool = False

while not done:
  done = True



  while True:
    r = find_neighbor(left, p, isClockwise=True)
    if r == p:
      break
    prev_slope = temp
    curr_slope = calculate_slope(r, q)

    if is_bigger_tangent_line(prev_slope, curr_slope):
      temp = curr_slope
      p = r
      done = False
    else:
      break



  while True:
    v = find_neighbor(right, q, isClockwise=False)
```

```
        if v == q:
            break
        prev_slope = temp
        curr_slope = calculate_slope(v, p)

        if is_smaller_tangent_line(prev_slope, curr_slope):
            temp = curr_slope
            q = v
            done = False
        else:
            break

    return (p, q)
```

O(log(n)) + O(log(n)) = O(log(n)), this will loop through parts of two loops where the average looping will not be of O(n)

```
def combine(upper:list[tuple], lower:list[tuple], left:list[tuple[float, float]],
        right:list[tuple[float, float]])-> list[tuple[float, float]]:
    '''
    Starting with Right's lower tangent point, iterate until through until you reach Right's upper tangent point.
    Then interate through Left's points starting at the upper tangent point until you get to the Right's lower tangent point
    '''
    #  idx = points.index(current_point)
    # if isClockwise:
    #     return points[(idx + 1) % len(points)]
```

```
    # else:
    #     return points[(idx - 1) % len(points)]
    combined_hull: list[tuple[float,float]] = []
```

O(log(n)), I am going to be consistent and say that that on average, we will have a O(log(n)) for finding a specific index

```
    right_start = right.index(lower[1])
    right_stop = right.index(upper[1])


    left_start = left.index(upper[0])
    left_stop = left.index(lower[0])
```

O(1)

```
    if len(right) == 1:
        combined_hull.append(right[0])
    else:
```

O(1)

```
        current_idx = right_start
```

O (log(n)), I say this because we probably will not be going through every point for the hull, but it is definitely isn't O (n) time

```
        while True:
```

O (1)

```
            combined_hull.append(right[current_idx])
```

O (1)

```
            if right[current_idx] == right[right_stop]:
                break
```

O (1)

```
        current_idx = (current_idx + 1) % len(right)


    if len(left) == 1:
        combined_hull.append(left[0])
    else:
        current_idx = left_start
            O(log(n)) -> see the previous for loop
        while True:


            combined_hull.append(left[current_idx])


            if left[current_idx] == left[left_stop]:
                break


            current_idx = (current_idx + 1) % len(left)


    return combined_hull
```

O(n) * O (log(n)) = O(n log(n))

```
def compute_hull_helper(points: list[tuple[float, float]]) -> list[tuple[float, float]]:
    """Return the subset of provided points that define the convex hull"""
```

O(1)

```
    theta = 2 #threshold value
```

O(1)

```
    if len(points) <= theta:
        return points
```

O(1), see above explanation

average_x = find_average_x(points)

O(1), just initializing lists

left:list[tuple[float, float]] = []

right:list[tuple[float, float]] = []

O(n), see above explanation

left, right = split_into_left_right(points, left, right, average_x)

left_hull:list[tuple[float, float]] = compute_hull_helper(left)

right_hull:list[tuple[float, float]] = compute_hull_helper(right)

O(log(n)), see above explanation

upper:list[tuple] = list(find_upper_Tangent(left_hull, right_hull))

lower:list[tuple] = list(find_Lower_Tangent(left_hull, right_hull))

O(log(n)), see above explanation

convex_hull:list[tuple[float, float]] = combine(upper, lower, left_hull, right_hull)

return convex_hull

O(n log(n))

def compute_hull(points: list[tuple[float, float]]) -> list[tuple[float, float]]:

O (n), has to look at all the points

points.sort(key=lambda point: point[0])

O(n log(n)), see above explanation

```
convex_hull = compute_hull_helper(points)


return convex_hull
```

Master Theorem Exploration:

a = 2, b = 2, d = 1

so our recurrence relation is as follows:

T(n) = 2T(n/2) + O(n^1)

And

1 = (2/2^1)

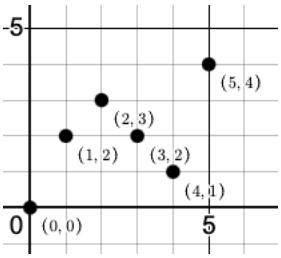
Thus,

O(n log(n))

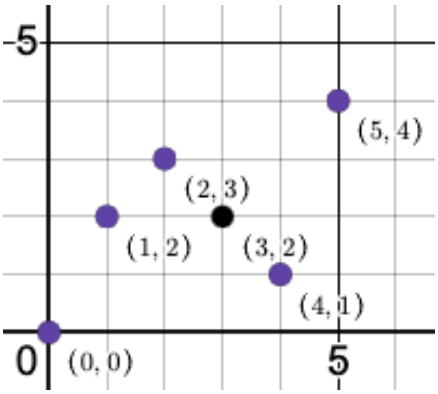
According to the Master theorem

Question 2

I created several small cases to check that my algorithm was working correctly. The one that was the most difficult to solve actually only ended up having six points. Here is what it looked like before I did the convex hull algorithm:



This was a fairly simple test where I was hoping to only get rid of one point from one side while I kept the rest of the points on the other. Here was the result after I got the algorithm working correctly:



The points that were included are the purple points.

Based on my calculations in problem 1, the growth is about O(nlog(n)).

If the constant of proportionality is a in the recurrence relation, I found that to be 2.

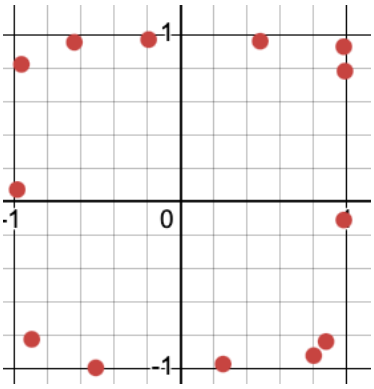
Some of the assumptions I made during the calculations were that the modulus operations were of constant time. I researched if that was a fair assumption, and for the most part it was a fair. I made the assumption that the rotation of the list in the “while” loops would average a time complexity of O(log n) because I figured that on average, we would not need to iterate through all the points, so it wouldn’t be O(n) time, but it also wouldn’t be O(1) constant time either due to the fact that we are iterating through values.

Question 3

My theoretical analysis proved to be very wanting. I looked at the pseudo code provided in class and thought I had an idea about what was going on. For example, in the Calc_Upper_Tangent function, I assumed I understood the code when it said “check if is not a tangent”. I didn’t realize at the time that I should be looking in a change increasing the slope/decreasing the slope as I rotated around the two different hulls. I feel like this is eye opening for me to understand how to look at pseudo code more analytically.

Question 4

Here is the hull after 100 points:



Here is what 1000 point looks like: