Part 1: Time and Space Complexity

Contributing Functions (ordered from most nested to least nested):

- notCalibrated() (all directions have same complexities)
- findUpper / findLowerTangent() (both have same complexities)
- combine()
- sorted()
- divide_and_conquer()
- makeResultList
- compute_hull()

Format: I will name the function as well as its space and time complexity. I will then include a picture of the code of the function along with the explanation of the time/space complexity. This may be easier read from the bottom up. Or from comput_hull back up to notCalibrated().

Function: notCalibrated()
Time: O(1)/O(constant)
Space: O(1)/O(constant)
Note: I have four versions of this function, one for each direction. Each has the same complexities.
Explanation:
Time complexity: Calculating the slope is constant, and comparison is also constant, so the time complexity is O(constant).
Space complexity: Two values are required to hold slopes, which is also constant. O(constant).
Reference:

```
def upperLeftNotCalibrated(lNode: Node | None, rNode: Node | None) -> bool:
    """
    Upper left is not calibrated as long as if we went counter clockwise, we got a lower slope
    """
    if lNode is not None and rNode is not None:
        initialSlope = calculate_slope(lNode.get_point(), rNode.get_point())
        potentialNewSlope = calculate_slope(
            lNode.get_lNode().get_point(),  # type: ignore
            rNode.get_point(),
        )
        return potentialNewSlope < initialSlope
    print("Node was None")
    return False
```

Function: findUpperTangent()
Time: O(n)
Space: O(constant)

Note: There is a similar function, findLowerTangent, with the same complexities
Explanation:
Time: At worst, the leftmost and rightmost points will be at the very bottom, and every node will have to be checked twice, making this O(2n), where n is the length of the left and right lists. So this will be linear: O(n).
Space: There is no recursion, and we only create a constant number of variables to hold the nodes, so this is constant space. Although we might change leftTangent or rightTangent n times, assuming we are just replacing the old variable, this is still constant.
Reference:

```python
def findUpperTangent(
    lList: list[Node], rList: list[Node]
) -> tuple[Node | None, Node | None]:
    leftTangent: Node | None = lList[len(lList) - 1]
    rightTangent: Node | None = rList[0]
    temp = (leftTangent, rightTangent)
    draw_line(rightTangent.get_point(), leftTangent.get_point())  # type: ignore

    done = False
    while not done:
        done = True
        while upperLeftNotCalibrated(temp[0], temp[1]):
            if leftTangent is not None:
                leftTangent = leftTangent.get_lNode()
                temp = (leftTangent, rightTangent)
                draw_line(leftTangent.get_point(), rightTangent.get_point())  # type: ignore
                done = False
        while upperRightNotCalibrated(temp[0], temp[1]):
            if rightTangent is not None:
                rightTangent = rightTangent.get_rNode()
                temp = (leftTangent, rightTangent)
                draw_line(leftTangent.get_point(), rightTangent.get_point())  # type: ignore
                done = False

    return temp
```

Function: combine()
Time: O(n)
Space: O(constant)
Explanation:
Calculating upper and lower tangents is linear time, O(n), and has constant space complexity.
Removing bad nodes is constant space and time, as it involves just reassigning pointers in the leftmost and rightmost nodes.
Overall time is O(n) + O(constant) = O(n).
Overall space is O(constant).
Reference:

```python
def combine(lList: list[Node], rList: list[Node]) -> list[Node]:
    # find upper tangent
    upperBound: tuple[Node | None, Node | None] = findUpperTangent(lList, rList)
    lowerBound: tuple[Node | None, Node | None] = findLowerTangent(lList, rList)
    # remove unnecesary edges
    removeBadNodeConnections(upperBound, lowerBound)
    # lList.extend(rList)
    newList = [lList[0], rList[len(rList) - 1]]
    return newList
```

Function: sorted()
Time: O(nlogn)
Space: O(n)
Note: built in python method used in compute_hull()

Function: divide_and_conquer
Time: O(nlogn)
Space: O(n)
Explanation:
Time: The algorithm creates two new problems of half the size. The combining operation takes O(n), so the overall recurrence equation is T(n) = 2T(n/2) + O(n). According to the Master Theorem, if we calculate a/b^d, we get 2/2^1, which equals 1. When this equals 1, the time complexity is T(n) = O(n^d log n). Since d is 1, we end up with O(n log n) for time complexity.

Space: The input is of size n, and we create a node for every input, resulting in n nodes. While we create new lists, those lists only contain a constant 2 elements. The recursion utilizes the stack, but since the problem is split in half with each call, the maximum depth reached is log base 2 of n. Thus, the space complexity for the stack is O(log n). Combining everything, we have n (input) + n (nodes created) + log n (stack storage). Since n dominates, the overall space complexity is O(n).

Reference:

```python
def divide_and_conquer(points: list[tuple[float, float]]) -> list[Node]:
    if len(points) == 1:
        return createNodes(points)
    else:
        # assert that this splits list into two halves
        lList = divide_and_conquer(points[: len(points) // 2])
        rList = divide_and_conquer(points[len(points) // 2 :])
        combinedList = combine(lList, rList)
        return combinedList
```

Function: makeResultList()
Time: O(n)
Space: O(n)
Explanation:
Time: Going through each node to insert into resList at worst takes O(n).
Space: At worst, a list of length n is created, so the space complexity is O(n).
Reference:

```python
def makeResultList(originNode: Node) -> list[tuple[float, float]]:
    resList: list[tuple[float, float]] = []
    resList.append(originNode.get_point())  # type: ignore
    nextNode = originNode.get_rNode()
    while nextNode != originNode and nextNode:
        resList.append(nextNode.get_point())
        nextNode = nextNode.get_rNode()
    return resList
```

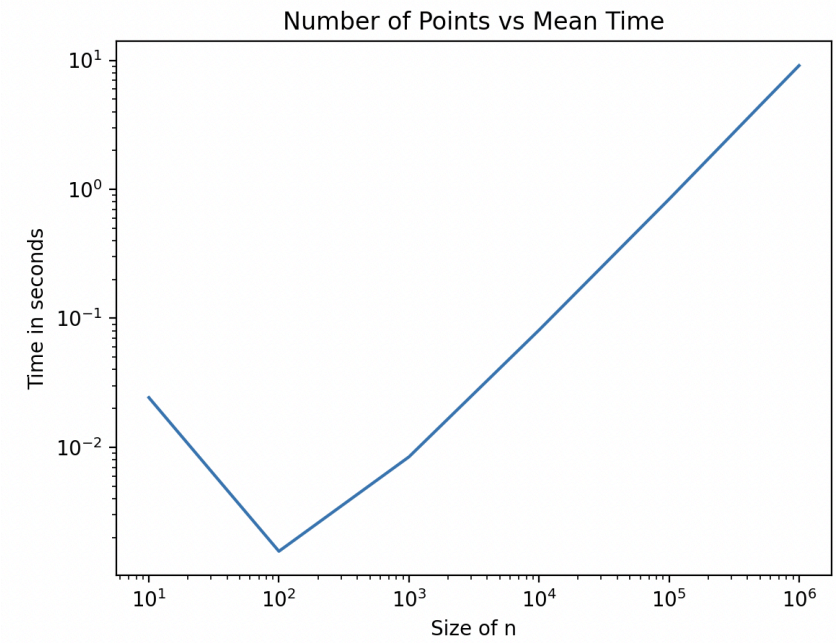
Function: compute_hull()
Time: O(nlogn)
Space: O(n)
Note: The combination of all other functions
Explanation:
Time: Sorting takes O(n log n). The divide_and_conquer function also takes O(n log n). Creating the result list takes O(n). The total is n log n + n log n + n = O(n log n).
Space: The input size is n. Python sorting at worst takes O(n) space. The divide_and_conquer function takes O(n) space as well. Creating the result list at worst takes O(n). The total is n + n + n + n = O(4n) = O(n).

Reference:

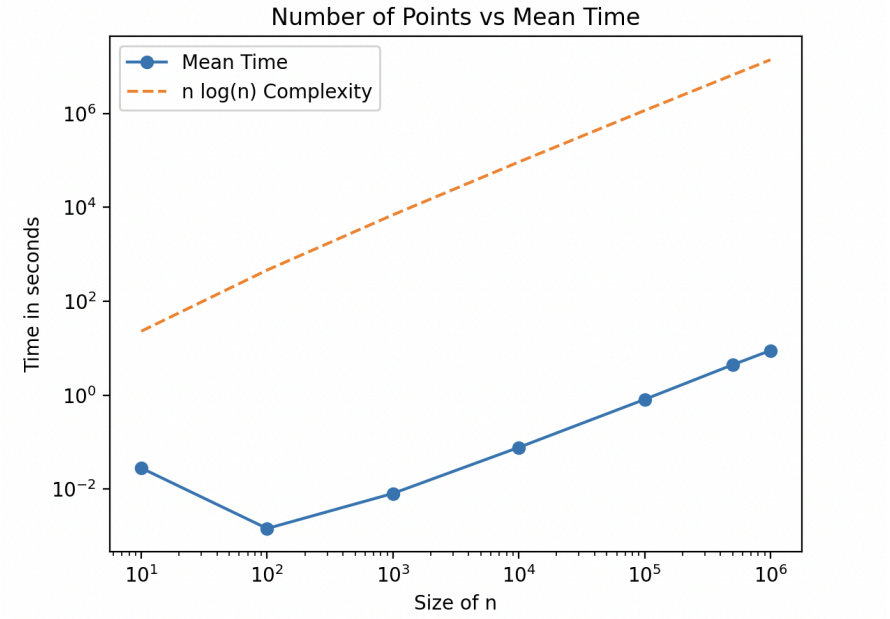
```python
def compute_hull(points: list[tuple[float, float]]) -> list[tuple[float, float]]:
    """Return the subset of provided points that define the convex hull"""

    points = sorted(
        points, key=lambda point: point[0]
    )  # sort hull points by x-coordinate
    plot_points(points)

    node_list: list[Node] = divide_and_conquer(points)
    resList: list[tuple[float, float]] = makeResultList(node_list[0])

    return resList
```

Part 2: Graphs, Data, and Discussions

First Graph:



Second Graph:



Raw Data:

10 points:
0.13463807106018066
0.0009639263153076172
0.0007889270782470703
0.0007309913635253906
0.0007212162017822266
100 points:
0.00140595436096191 4
0.0013968944549560547
0.0015511512756347656
0.00138092041015625
0.0013859272003173828
1000 points:
0.008278846740722656
0.00848698616027832
0.008132219314575195
0.008217096328735352
0.00818490982055664
10000 points:
0.07802581787109375
0.07733011245727539
0.0770418643951416
0.07747507095336914
0.0776817798614502
100000 points:
0.8028500080108643
0.8283779621124268
0.8036210536956787
0.8086428642272949
0.8066301345825195
500000 points:
4.372117042541504
4.435017824172974
4.419079065322876
4.364722967147827
4.428416967391968
1000000 points:
9.022086143493652
8.897279024124146
8.89484691619873
8.900553941726685
8.994270086288452

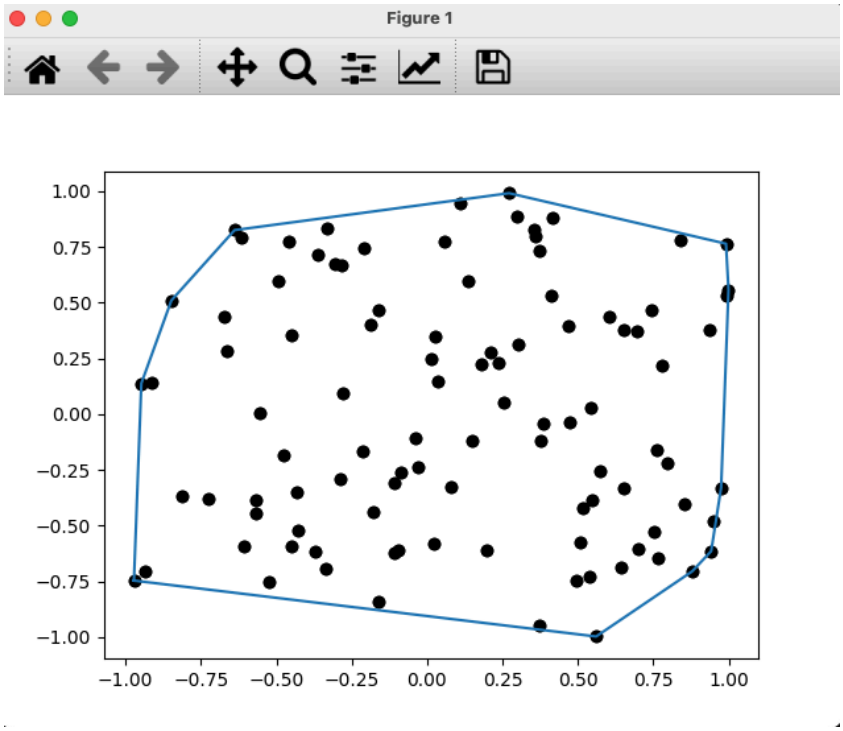
Average Data:
For 10 points, the average time taken was 0.02757 seconds.
For 100 points, the average time taken was 0.00142 seconds.
For 1,000 points, the average time taken was 0.00826 seconds.
For 10,000 points, the average time taken was 0.07751 seconds.
For 100,000 points, the average time taken was 0.81002 seconds.
For 500,000 points, the average time taken was 4.40387 seconds.
For 1,000,000 points, the average time taken was 8.94181 seconds.

Pattern of Plot:
Although off by a multiple of a constant, my graph fits the expected nlogn time complexity that it should theoretically fit. The easiest way to compute the constant of proportionality was to compare it to the graph of a regular nlogn line. Analyzing the numbers, it looks to be off by a constant of 10^6. This was done by comparing values on the graph to each other.

Theoretical vs Empirical Data:

The graphs appear to be very similar. Although they are off by that constant of proportionality, the slopes are very similar. One big difference is when the size of n is equal to ten. Although it may appear as if the empirical data gets faster from 10-1000 points, this is not so. If we take a deeper look at the individual data points rather than the mean, we see that the very first time this program runs, it takes 0.13463807106018066 seconds. After that, the average goes down tremendously to about 0.0007889270782470703 seconds. This first data point completely skews that data and is most likely due to how python compiles/runs its programs. Ignoring this outlier, the empirical data has an extremely similar slope to the theoretical data.

Hull with 100 points:



Hull with 1000 points: