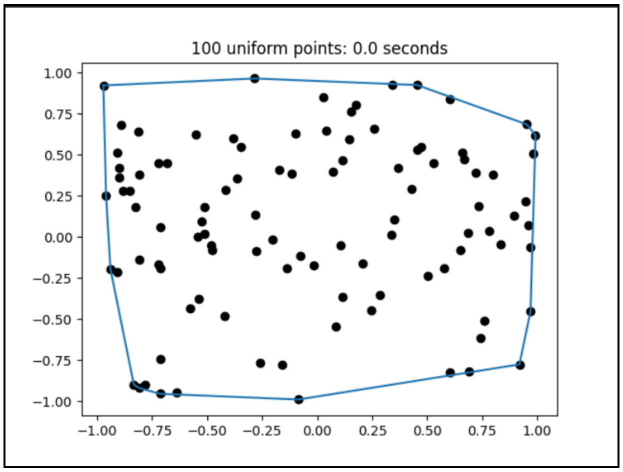
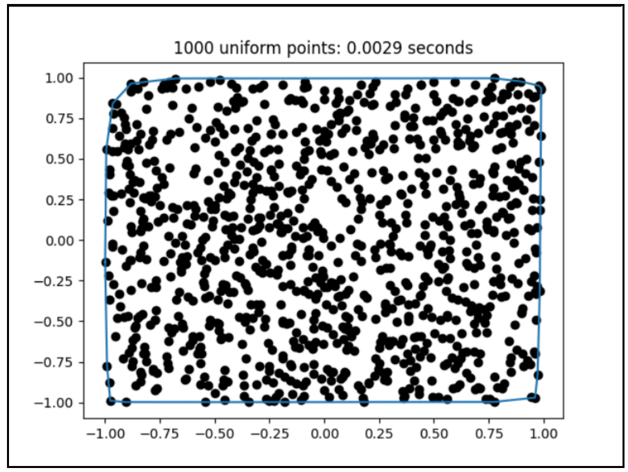
Project 2 Report





Complexity Analysis

Before I could fully implement the convex hull algorithm, I needed to create some data structures and functions to help.

Node

```
class Node:  11 usages
    def __init__(self, coordinates:tuple[float, float]):
        self.coordinates = coordinates
        self.clockwise = None
        self.counterclockwise = None

    def add_clockwise(self, node: 'Node'):  2 usages (1 dynam
        self.clockwise = node
        node.counterclockwise = self

    def add_counterclockwise(self, node: 'Node'):  2 usages
        self.counterclockwise = node
        node.clockwise = self
```

The Node class is very simple. It contains the coordinates to whatever point it corresponds with and points to the clockwise and counterclockwise nodes. It also has two functions to set the clockwise or counterclockwise nodes to it. All three of these functions can be done in constant time.

Slope

```
def slope(left: Node, right: Node) -> float:  6 usages
    return ((right.coordinates[1] - left.coordinates[1])
            / (right.coordinates[0] - left.coordinates[0]))
```

The slope function is also very simple. Because all it does are basic operations (subtract, division, indexing), the function is able to calculate the slope between two nodes in constant time.

Hull

The Hull class is where things get a little bit more complicated. However, initializing a Hull is pretty simple. It takes the coordinates it receives, creates a node from those coordinates, and assigns it to "leftmost" and "rightmost", all in constant time. The function "join_two_nodes" is able to join itself with a hull to the right, provided that both hulls only have one node contained in them. It is able to do this all in constant time.

```
class Hull:  7 usages
    def __init__(self, coordinates:tuple[float, float]):
        node: Node = Node(coordinates)
        node.clockwise = node
        node.counterclockwise = node
        self.rightmost = node
        self.leftmost = node

    def join_two_nodes(self, right_hull: 'Hull'):  1 usage
        """Only use when joining two hulls that both only have one
        self.rightmost.add_clockwise(right_hull.leftmost)
        self.rightmost.add_counterclockwise(right_hull.leftmost)
        self.rightmost = right_hull.rightmost
```

The function "set_upper_tangent" finds the two points that connect the upper tangent of the left and right hulls. It does this by traveling counterclockwise from the rightmost node of the left hull and clockwise from the leftmost node of the right hull until it finds the upper tangent and returns the two nodes that need to be connected later on. Assuming that the hulls are of roughly equal

```
def set_upper_tangent(self, right_hull: 'Hull') -> (Node, Node):
    l = self.rightmost
    r = right_hull.leftmost
    tan = slope(l,r)
    done = False
    while not done:
        done = True
        while True:
            if l.counterclockwise == self.rightmost:
                break
            temp = slope(l.counterclockwise, r)
            if temp < tan:
                tan = temp
                l = l.counterclockwise
                done = False
            else:
                break
        while True:
            if r.clockwise == right_hull.leftmost:
                break
            temp = slope(l, r.clockwise)
            if temp > tan:
                tan = temp
                r = r.clockwise
                done = False
            else:
                break
    return l, r
```

size, and that the nodes are evenly distributed, this function would probably only traverse a quarter of the nodes in each hull on average. Therefore, this function's time complexity would be linear, or in the order of O(n). Even in the worst case, it would only traverse at most n-1 nodes, thus remaining O(n). The function "set_lower_tangent" does the exact same thing but with the lower tangent of the two hulls instead. Because of this, its time complexity would also be linear, or in the order of O(n).

```
def set_lower_tangent(self, right_hull: 'Hull') -> (Node, Node):
    l = self.rightmost
    r = right_hull.leftmost
    tan = slope(l,r)
    done = False
    while not done:
        done = True
        while True:
            if l.clockwise == self.rightmost:
                break
            temp = slope(l.clockwise, r)
            if temp > tan:
                tan = temp
                l = l.clockwise
                done = False
            else:
                break
        while True:
            if r.counterclockwise == right_hull.leftmost:
                break
            temp = slope(l, r.counterclockwise)
            if temp < tan:
                tan = temp
                r = r.counterclockwise
                done = False
            else:
                break
    return l, r
```

```
def hull_join(self, right_hull: 'Hull'):  1 usage
    ul, ur = self.set_upper_tangent(right_hull)
    ll, lr = self.set_lower_tangent(right_hull)
    ul.add_clockwise(ur)
    ll.add_counterclockwise(lr)
    self.rightmost = right_hull.rightmost
```

The function "hull_join" uses the points returned from the previous functions to join two hulls that have more than one node each. Making the same assumptions as before, this function would only have to traverse around half of each hull (at most n-1 nodes) and can 'throw away' all the nodes it traverses before finding the upper and lower tangents. Thus, this function can also be performed in linear time and is in the order of O(n).

Convex Hull Algorithm

```
def recursive_hull(points: list[tuple[float, float]]) -> Hull:
    if len(points) == 1:
        return Hull(points[0])
    if len(points) == 2:
        left_hull = recursive_hull([points[0]])
        right_hull = recursive_hull([points[1]])
        left_hull.join_two_nodes(right_hull)
        return left_hull
    else:
        mid = len(points) // 2
        left_hull = recursive_hull(points[:mid])
        right_hull = recursive_hull(points[mid:])
        left_hull.hull_join(right_hull)
        return left_hull
```

The “recursive_hull” function does most of the heavy lifting by combining the functionality of everything previously discussed. It is a recursive function, with 2 base cases. If there is only 1 coordinate in the list, it simply creates a Hull in constant time. If there are 2, it gets the hulls for each coordinate and performs “join_two_nodes”, joining the 2 hulls in constant time. Otherwise, it splits the list into 2 even lists and recursively calls itself using those 2 lists. It then performs “hull_join” on the 2 hulls returned from the recursive calls. We can use the Master theorem to determine the time complexity of this function. The branching factor is 2, so a = 2. The pre/post-work complexity is linear O(n), so d = 1. The reduction factor is n/2, so b = 2. With all of that in mind, a/(b^d) = 1, so the time complexity of this function is O(n*log(n)).

The “compute_hull” function wraps the previous function and extracts the list of coordinates from the hull. Since “recursive_hull” is the most complex part of this function, it also runs in O(n*log(n)) time.

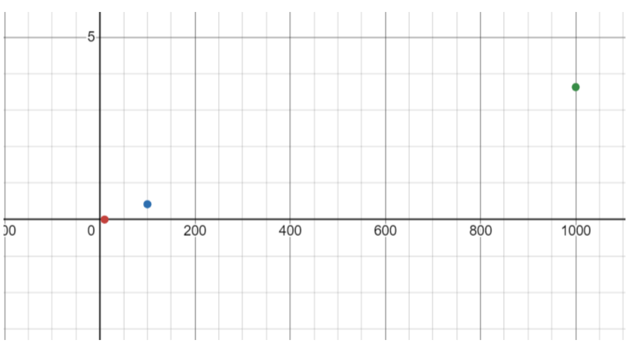
```
def compute_hull(points: list[tuple[float, float]]) -> list[tuple[float, float]]:
    """Return the subset of provided points that define the convex hull"""
    sorted_points = sorted(points, key=lambda point: point[0])
    hull: Hull = recursive_hull(sorted_points)
    hull_list: list[tuple[float, float]] = []
    hull_node: Node = hull.leftmost
    hull_list.append(hull_node.coordinates)
    while True:
        hull_node = hull_node.clockwise
        if hull_node == hull.leftmost:
            break
        else:
            hull_list.append(hull_node.coordinates)
    return hull_list
```

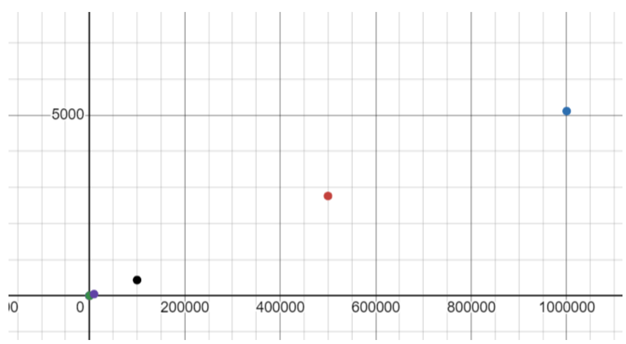
Empirical Analysis

The following is a list of times taken to compute convex hulls of differing sizes and their averages.

n	10	100	1,000	10,000	100,000	500,000	1,000,000
t1	0 ms	0 ms	5 ms	49.7 ms	465.5 ms	2407 ms	4441 ms
t2	0 ms	0 ms	4.1 ms	31.5 ms	537.6 ms	2288 ms	4362 ms
t3	0 ms	0 ms	3 ms	74.2 ms	402.6 ms	3081 ms	4370 ms
t4	0 ms	1 ms	3 ms	42.8 ms	396.3 ms	3774 ms	7137 ms
t5	0 ms	1.1 ms	3.1 ms	32.4 ms	377.1 ms	2267 ms	5245 ms
t-avg	0 ms	0.42 ms	3.64 ms	46.12 ms	435.8 ms	2763 ms	5111 ms

Here are some plots for those points on a graph. The first graph corresponds to the first 3 n-values from 10 to 1,000 and the second graph corresponds to the last 5 n-values from 1,000 to 1,000,000.





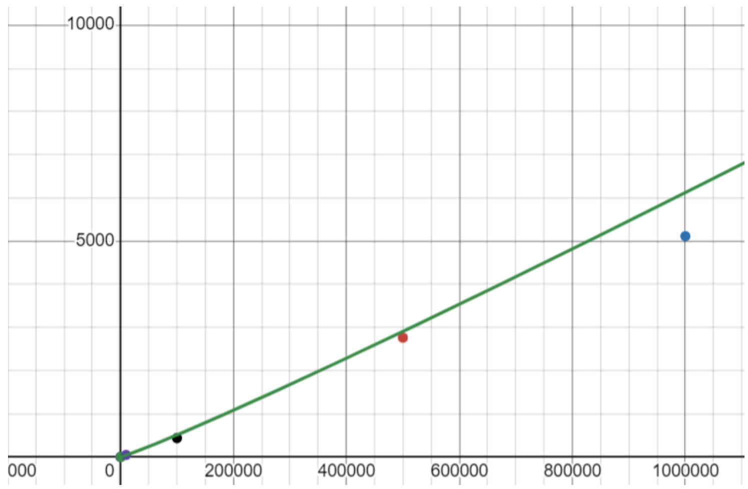
To find the constant of proportionality we can use our theoretical n*log(n) complexity and the results of our empirical data in the form of the following equation: t = k*(n*log(n)). Solving for k gives us the following equation: k = t / (n*log(n)). Using that equation gives us the following values for k.

n	10	100	1,000	10,000	100,000	500,000	1,000,000
k	0	0.0021	0.0012	0.0012	0.00087	0.00097	0.00085

Finding an average k-value can give us a good constant of proportionality. Calculating the average of our values for k gives us 0.00102. Using this gives us an equation of

t = 0.00102(n*log(n)).

Plotting that equation shows that our empirical results do fall under the line.



Conclusion

I find it interesting that there is some variability in the observed k-values, suggesting that, in reality, running the algorithm may not exactly line up with one particular equation. However, that can also be chalked up to the varying speeds of my computer when I run the algorithm at different times. I'm sure that if I were to gather more empirical data, the k-values for those results could be a little bit different than what has already been observed. Thankfully, being able to show that there is a n*log(n) equation where all the points fit under it is enough to show that the algorithm really does run in O(n*log(n)) time.