In this project, we compared the efficiencies of a brute force algorithm versus a recursive algorithm when it comes to finding the distance between the 2 closest points in the data set.

This report will be organized in the following pattern: first, a theoretical expression of the efficiencies of the different programs, followed by time trials proving the correctness of the theory expressed before it. Firstly, the time complexity of the brute force algorithm will be determined:

Code	Times
`def brute_force(point_set):`	1
`    num_points = len(point_set)`	1
`    shortest_distance = sys.maxsize - 1`	1
`    point1 = None`	1
`    point2 = None`	1
`    for i in range(num_points - 1):`	$n-1$
`        for j in range(i + 1, num_points):`	$\sum_{i=1}^{n-1} i$
`            distance = math.sqrt((point_set[i][0] - point_set[j][0]) ** 2 + (point_set[i][1] - point_set[j][1]) ** 2)`	$\sum_{i=1}^{n-2} i$
`        if distance < shortest_distance:`	$\sum_{i=1}^{n-2} i$
`            shortest_distance = distance`	$\sum_{i=1}^{n-2} i$
`            point1 = point_set[i]`	$\sum_{i=1}^{n-2} i$
`            point2 = point_set[j]`	$\sum_{i=1}^{n-2} i$
`    return shortest_distance`	1

As it can be seen from the code above, it is clear that the most complex lines of code in the project are the original *for* loop (complexity $n - 1$) and the inner for loop (complexity $\sum_{i=1}^{n-1} i$). Therefore, if we were to set up and solve the equation for the time complexity family for the brute force function, it would be as follows:

$$T(n) = n - 1 + \sum_{i=1}^{n-1} i + K$$

It should be noted that the constant K represents the time complexity for all of the other parts of the algorithm. However, because ever other part of the algorithm grows slower than the first parts of the equation, they are insignificant to the final solution.

$$T(n) = n - 1 + \frac{(n-1)\,n}{2}$$
$$T(n) = n - 1 + \frac{n^2 - n}{2}$$
$$T(n) = O(n^2)$$

Therefore, it has been proved that the time complexity family for the brute force function is $O(n^2)$. Now, a similar method will be employed to determine the complexity family for the recursive function.

```
# Recursive function. Takes in 2 lists: one with the points sorted with respect to x
#     and one with the points sorted with respect to y
def efficient_closest_pair(p, q):
    if len(p) <= 3:
        return brute_force(p)

    else:
        pl = p[:int(len(p)/2)]
        ql = q[:int(len(q)/2)]
        pr = p[int(len(p)/2):]
        qr = p[int(len(q)/2):]

        dl = efficient_closest_pair(pl, ql)
        dr = efficient_closest_pair(pr, qr)

        d = min(dl, dr)
        m = p[int(math.ceil(len(p)/2)) - 1][0]

        s = []
        for point in q:
            if abs(point[0] - m < d):
                s.append(point)

        dminsq = d ** 2

        for i in range(len(s) - 1):
            k = i + 1

            while k < len(s) and (s[k][1] - s[i][1]) ** 2 < dminsq:
                dminsq = min((s[k][0] - s[i][0]) ** 2
                        + (s[k][1] - s[i][1]) ** 2,
                        dminsq)
                k += 1

        return math.sqrt(dminsq)
```

However, for this algorithm, we won't delve into as much detail as the previous algorithm. This is simply because most of the algorithm is very obviously either O(1) or O(n), time, and as it will be shown later in the paper, that will be insignificant in the final equation.

The mains points in interest in this algorithm is that there are 2 recursive calls, each of size $\frac{n}{2}$.

Additionally, there is a nested while & for loop that will change based on the best and worst cases. The time count for the while loop is $\sum_{i}^{n-1} t_i$. The value of t_i will change based on the data inputted.

Best Case:

If the data is ordered in such a way that the shortest distance was determined in the last recursive call, then that would mean that the condition for the while loop would always fail, as `(s[k][1] - s[i][1]) ** 2` would always be greater than `dminsq`. That means that the recurrence equation would become:

$$T(n) = 2T\left(\frac{n}{2}\right) + O\left(\sum_{i}^{n-1} 1\right)$$
$$T(n) = 2T\left(\frac{n}{2}\right) + O(n-1)$$
$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Then, in accordance to the master method:

$$O\left(n^{\log_2 2}\right) vs. O(n)$$
$$O(n) vs. O(n)$$
$$T(n) = \Omega(n \log n)$$

Worst Case

Assume that the shortest distance was the last 2 points that were compared. That would mean that that `s[k][1] - s[i][1]) ** 2` would always be less than `dminsq`. Then, the recurrence equation would become:

$$T(n) = 2T\left(\frac{n}{2}\right) + O\left(\sum_{i}^{n-1} i\right)$$

$$T(n) = 2T\left(\frac{n}{2}\right) + O\left(\frac{(n-1)\,n}{2}\right)$$

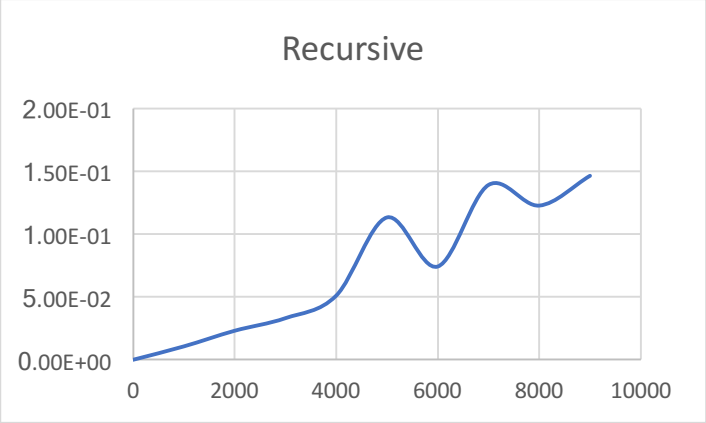
$$T(n) = 2T\left(\frac{n}{2}\right) + O(n^2)$$

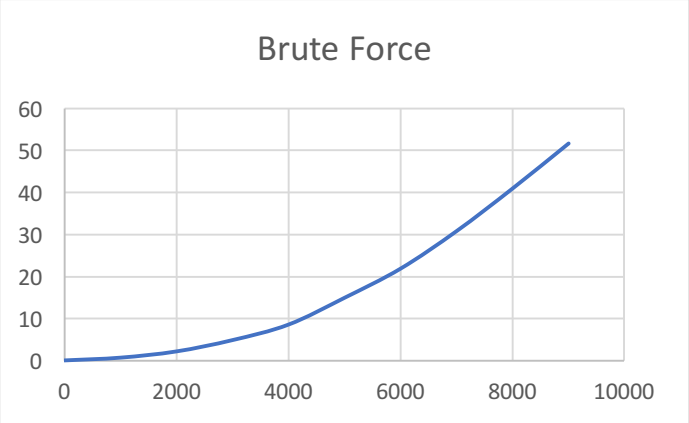
Again, following the master method:

$$O\left(n^{\log_2 2}\right) vs. O(n^2)$$

$$T(n) = O(n^2)$$

Thus, it has been proven that the time complexity of the recurrence function will be bounded by $nlogn \leq T(n) \leq n^2$. Now as shown below, here is the running data graphed n = [0, 900]:





As you can see, the Brute force method is clearly n^2, and the recursive function seems to grow slower than n^2, but with a lower bound of $n \log n$, which is in accordance with what was proven above.