Part 1 (Jarvis march algorithm)

Computational Complexity:

Random input case:

Theoretically: The total run time is O(Nh), where N is the size of input data, h is the size of the output(the subset of points that lie on the convex hull). $h \leq N$

O(Nh), h depends on how the function ' random _data' define.

```
def random_data(N):
    inputlista = [[random.randint(0, 32767) for j in range(1, 3)] for i in range(N)] #get N pairs random points
    return inputlista
```

Experimentally:

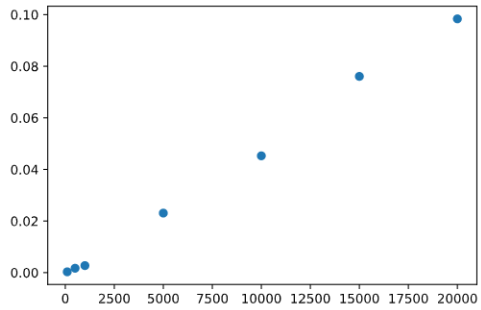
N = 100, 500, 1000, 5000, 10000,15000,20000

h = 10,15,19,25,19,20,32

(note: We run the test 100 times with each N to reduce errors, and divide each total time of N by 100 to get average execution time)

execution time (s) = [0.0003140000001167209, 0.0017018999999436346, 0.00272610000001805, 0.023069800000030227, 0.04527290000009998, 0.07606239999995523, 0.09836279999990438]

Scatter plots (x-axis is N , y-axis is the measured execution time):



Conclusion:

With the range of x-y scale of input points unchanged ([0,32767]) , the value of h does not change much when N takes 100,500,1000,50000,10000,15000,20000. The value of h is much less than N, and h is not proportional to N.

The worst-case:

Theoretically: h = N →O(N^2)

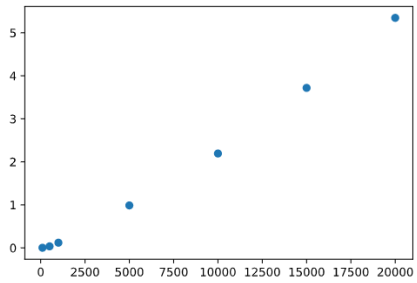
Experimentally: ' h = N ' means all of the input points lie on the convex hull. The code used here is an example which is a circle with center (32767/2 ,32767/2).

'input' N = 100,500,1000,5000,10000,15000,20000

Note: The actual value of N is always less than the 'input ' N. This is because we use ' random ' . It causes the larger 'input' N , the more repetitions. But in the actual algorithm, there is no repeated point by default.

execution time(s) = [0.002981100000170045, 0.03568360000008397, 0.12046559999998863, 0.9870078999999805, 2.193548299999975, 3.7183849999998984, 5.345930099999805]

Scatter plots (x-axis is N , y-axis is the measured execution time):



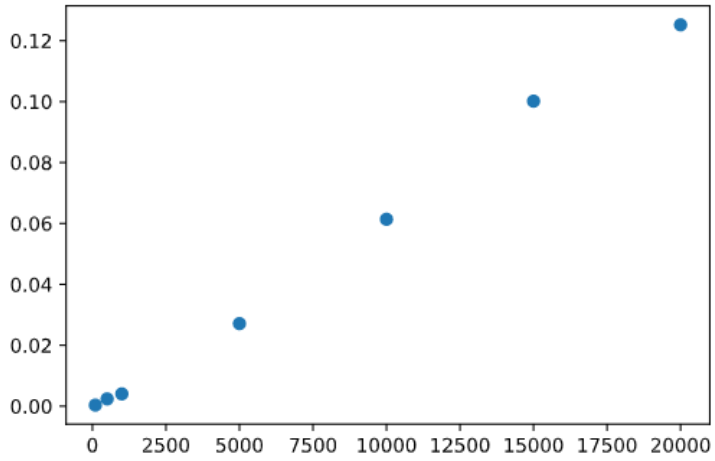
Part 2(Graham Scan algorithm)
Computational Complexity:

Random input case:

Theoretically:
There are 3 phases of this algorithm.
Phase 1 is getting the point p0 with minimum y-coordinate, or the leftmost such point in case of a tie. This takes time O(N).
Phase 2 is sorting the point by polar angle in counterclockwise order around p0, this should take time O(NlogN).

```
def quicksort(points, anchor):
    if len(points) <= 1:
        return points
    smaller, equal, larger = [],[],[]
    pivot = polar_angle(points[randint(0,len(points)-1)], anchor)
    for point in points:
        angle = polar_angle(point, anchor)
        if angle < pivot:
            smaller.append(point)
        elif angle == pivot:
            equal.append(point)
        else:
            larger.append(point)
    return quicksort(smaller, anchor) \
          + sorted(equal, key = lambda x: distance(x,anchor)) \
          + quicksort(larger, anchor)
```

Phase 3 goes through every point, each point is pushed in the stack exactly once, and popped at most once, so this gives us O(N) time complexity.
The total run time is O(NlogN).
Experimentally:
N = 100, 500, 1000, 5000, 10000,15000,20000
execution time (s) = [0.0003669000000172673, 0.0023853000000144675, 0.004041199999960554, 0.027114400000073147, 0.06137509999996382, 0.10015129999999317, 0.1252054000000271]
Scatter plots (x-axis is N , y-axis is the measured execution time):



Worst case:

Theoretically:

Since the degree of the polynomial is determined by the term with the highest degree in the polynomial, the worst case of graham scan algorithm is determined by the worst case of sorting algorithm which has complexity of O(NlogN). The sort algorithm we use is quicksort, the worst-case scenario of quicksort is O(N^2) which can be achieved by selecting the "bad" pivot every time, like the minimum value in a sorted list. However, we enhanced quicksort algorithm in this problem with optimization of randomly choosing the pivot, this already ensures the theoretically best performance – it's highly unlikely for the machine-determined random function to encounter with the corresponding input set that gives us worst performance. So theoretically the complexity of worst case is still O(Nlogn); The O(N^2) case is reachable, but with the possibility of 0.

Experimentally:

Since in most situation it's still O(NlogN), it won't make much difference than previous. However, we manually create a situation that has O(N^2) complexity, by making the inputSet already sorted and the quick sort algorithm choosing the "bad pivot" every time.

```
def quicksort_worst_scenario(points, anchor):
    if len(points) <= 1:
        return points
    smaller, equal, larger = [],[],[]
    pivot = polar_angle(points[0], anchor)
    for point in points:
        angle = polar_angle(point, anchor)
        if angle < pivot:
            smaller.append(point)
        elif angle == pivot:
            equal.append(point)
        else:
            larger.append(point)
    return quicksort_worst_scenario(smaller, anchor) \
           + sorted(equal, key = lambda x: distance(x,anchor)) \
           + quicksort_worst_scenario(larger, anchor)
```

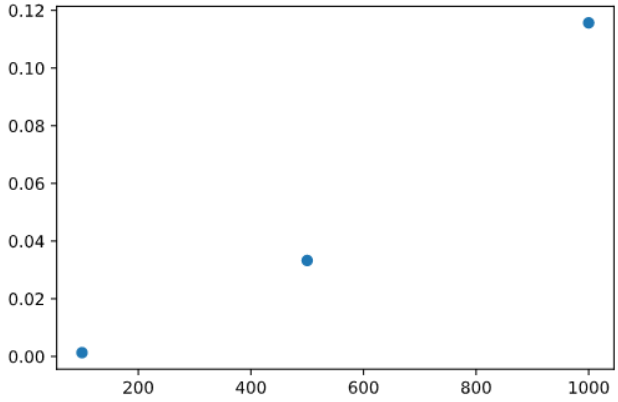
```
#Just one example
def worst_case_data(N):
    n = N
    inputlistb = []
    inputlistb.append([n//2,0])
    for i in range(1,n):
        inputlistb.append([i,n//2])
    return inputlistb
```

N = 100, 500, 1000, 5000, 10000,15000,20000

execution time (s) = [0.0012981000000991116, 0.03325760000006994, 0.11567089999994096, error, error, error, error]

Due to the limitation of maximum recursion depth in python (python doc, 2020) which has

the number of 1000, RecursionErrors will be reported for test sets with more than 1000 points because quick sort is a recursive approach – in this worst case, it calls the stack n times if there's n points in the test set . But we can see in the graph that the time worst case with 1000 points is nearly equal to average case with 15000 inputs.



Reference list:

"Python 3.9.2 documentation", Python Software Foundation. (2021). Available from: https://docs.python.org/3.9/