A-1.
Define your algorithm using pseudocode.

This algorithm calculates the ***m*** closest pairs of points from a given list of ***n*** points in two-dimensional space. The algorithm uses a divide and conquer approach to solve the problem, and a brute force method is used when the number of points is small or the number of pairs is close to the maximum number of pairs ***m***.

The algorithm works as follows:

1. The ***closest*** function that sorts the list of points by x coordinate,and calls the ***closest_recursive*** function.
2. The ***closest_recursive*** function recursively divides the set of points into two halves and combines the results of the two halves using the ***merge_deq*** and ***strip_scan*** functions. If the number of points is small or the number of pairs is close to the maximum number of pairs ***m***, the ***brute_force*** function is called.
3. The ***merge_deq*** function merges two DEPQs (double-ended priority queues) into one sorted DEPQ.
4. The ***brute_force*** function calculates the ***m*** closest pairs of points by comparing every pair of points and keeping track of the ***m*** smallest distances using a DEPQ.
5. The ***strip_scan*** function takes a list of points and a DEPQ and scans the points to find the closest pairs within a strip of width ***2d*** around the median line, where ***d*** is the largest distance of the ***m*** closest pair found so far.

The main data structure used in the algorithm is the DEPQ, which is a double-ended priority queue. The DEPQ is used to keep track of the m smallest distances found so far. The ***strip_scan*** function is used to find the closest pairs within a strip of points around the median line. The ***closest_recursive*** function recursively divides the set of points into two halves and combines the results of the two halves using the ***merge_deq*** and ***strip_scan*** functions. Finally, the closest function sorts the list of points by x coordinate, checks that there are enough pairs to find the ***m*** closest pairs, and calls the ***closest_recursive*** function.

A-2.

Determine the worst-case running time of my algorithm.

The worst-case running time of the algorithm can be determined by analyzing the time complexity of each of its components.

- The ***brute_force*** function has a time complexity of $O(n^2 \, logn)$
- The ***merge_deq*** function has a time complexity of $O(m \, logm)$
- The ***strip_scan*** function operates utmost 7 times for each points thus having a time complexity of $O(n)$
- The ***closest_recursive*** function has a time complexity of $T(n) = 2T(n/2) + O(m \, logm) + O(n)$, which by using the Master Theorem can be simplified to $O(n^2 \, logn)$ when $m = C\binom{n}{2}$.
- The ***closest*** function first sorts the list of points, which has a time complexity of $O(n \, logn)$, and then calls ***closest_recursive***, which has a time complexity of $O(n^2 \, logn)$.

Therefore, the overall worst-case running time of the algorithm is $O(n^2 \, logn) + O(n^2 \, logn) = O(n^2 \, logn)$.

B.

As shown in this GitHub repository

https://github.com/TimAgro/JHU_Algorithms/tree/main/Prog1

C.

Perform and submit trace runs demonstrating the proper function of you code

```
> python test/test_trace_run.py

These are the input points:
Point(x=2, y=3)
Point(x=12, y=30)
Point(x=40, y=50)
Point(x=5, y=1)
Point(x=12, y=10)
Point(x=3, y=4)

Enter the number of closest pairs you want to find:
1

This are the closest 1 pairs of points with its distance
(Point(x=2, y=3), Point(x=3, y=4)) | distance: 1.4142135623730951
```

```
> python test/test_trace_run.py

These are the input points:
Point(x=2, y=3)
Point(x=12, y=30)
Point(x=40, y=50)
Point(x=5, y=1)
Point(x=12, y=10)
Point(x=3, y=4)

Enter the number of closest pairs you want to find:
3

This are the closest 3 pairs of points with its distance
(Point(x=3, y=4), Point(x=5, y=1)) | distance: 3.605551275463989
(Point(x=2, y=3), Point(x=5, y=1)) | distance: 3.605551275463989
(Point(x=2, y=3), Point(x=3, y=4)) | distance: 1.4142135623730951
```

D.

Perform tests to measure the asymptotic behavior of your program.

Below is the performance table by executing this command:
> ***python test/test_performance.py***

execution time	*n*			
m	50	100	150	200
10	0.0023	0.0048	0.0051	0.0094
510	0.0088	0.0207	0.0441	0.0441
1010	0.0230	0.0402	0.0401	0.0847
1510		0.0253	0.0579	0.0598
2010		0.0330	0.0771	0.0778
2510		0.0408	0.0984	0.0958
3010		0.0484	0.0499	0.1164

3510		0.0580	0.0578	0.1380
4010		0.0665	0.0659	0.1602
4510		0.0760	0.0741	0.1804
5010			0.0841	0.0857
5510			0.0915	0.0952
6010			0.1018	0.1035
6510			0.1123	0.1113
7010			0.1242	0.1201
7510			0.1359	0.1312
8010			0.1506	0.1434
8510			0.1579	0.1506
9010			0.1688	0.1599
9510			0.1790	0.1685
10010			0.1939	0.1782
10510			0.2042	0.1917
11010			0.2181	0.2057
11510				0.2188
12010				0.2311
12510				0.2483
13010				0.2949
13510				0.2722
14010				0.2841
14510				0.2987
15010				0.3223
15510				0.3315
16010				0.3465
16510				0.3620
17010				0.3755
17510				0.3915
18010				0.4083
18510				0.4235
19010				0.4407
19510				0.4560

E.

Analysis comparing your algorithm's worst-case running time to your code's worst-case running time.

Looking at the table, we see that the input size is given by two parameters: n and m. Surprisingly, as shown in fig.1, the performance follows more by ***m*** than ***n***, which indicate that my algorithm is likely to behave with an $O(m)$ time complexity.

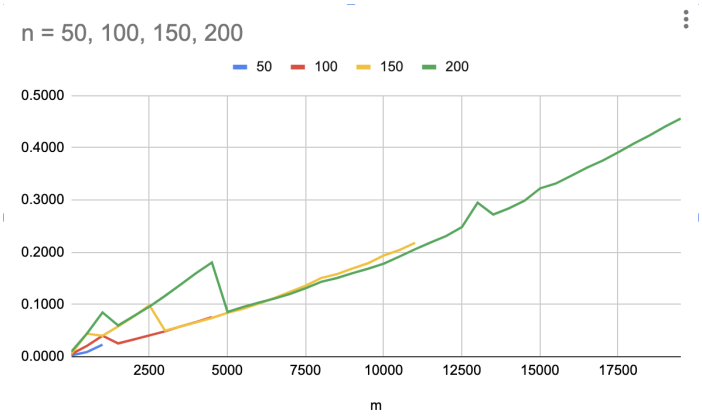

fig. 1

To simplify the analysis, we can assume that the input size is simply ***n***, and ignore the effect of m on the running time. From the plot, we see that the execution time appears to grow approximately quadratically with ***n***. However, there is some variation in the growth rate, particularly for larger values of ***n***.

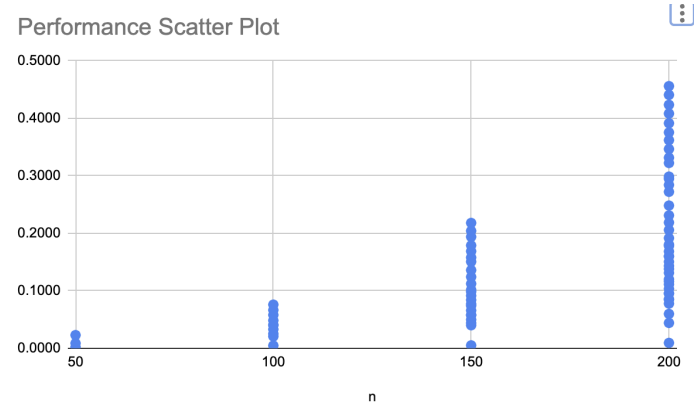

fig. 2

If we look back to the table, the rate of increase is not consistent. For example, when ***n*** = 50 and ***m*** increases from 10 to 1010, the execution time increases from 0.002 to 0.023, which is a factor of about 10. When ***n*** = 150 and ***m*** increases from 10 to 1010, the execution time increases from 0.005 to 0.04, which is a factor of about 8. However, when ***n*** = 200 and ***m*** increases from 10 to 1010, the execution time increases from 0.009 to 0.085, which is a factor of almost 10.

Therefore, we can conclude that the worst-case running time of the algorithm is at least $O(nm)$, but it may be higher depending on the specific values of ***n*** and ***m***.

Retrosprection

The use of dummy points in the ***brute_force*** function to ensure that the length of the result deq could equal ***m*** is actually redundant.

This is because if the number of points in the input is less than ***m***, then the size of the result deq will be less than ***m*** as well, regardless of whether or not we insert dummy points. On the other hand, if the number of points in the input is greater than or equal to ***m***, then the size of the result deq will be exactly m after the loop is finished, since we remove the highest distance points from the deq as necessary.

Therefore, we can remove the use of dummy points in the ***brute_force*** function without affecting its correctness.