2(a)i.

```
FIND-M-CLOSEST(P, m)
n = P.length
D = new array[1..n(n-1)/2]
k = 1
i = 1
while i < n
j = i + 1
while j ≤ n
dx = P[i].x - P[j].x
dy = P[i].y - P[j].y
distance = sqrt(dx * dx + dy * dy)
D[k].dist = distance
D[k].p1 = P[i]
D[k].p2 = P[j]
k = k + 1
j = j + 1
i = i + 1

MERGE-SORT(D, 1, k-1)

R = new array[1..m]
i = 1
while i ≤ m
    R[i] = D[i]
    i = i + 1

return R
```

2(a)ii.

The nested while loops execute for all unique pairs of points. The outer loop runs n-1 times and for each iteration i, the inner loop runs (n-i) times. This gives us: $\sum$(i=1 to n-1) (n-i) = (n-1) + (n-2) + ... + 1 = n(n-1)/2

Each iteration performs constant time operations, so the loops take $O(n^2)$ time.

MERGE-SORT on n(n-1)/2 elements takes $O(n^2 \log n^2) = O(n^2 \log n)$ time.

The final loop to build R runs m times with O(1) work per iteration, taking O(m) time.

Total worst-case time complexity: $O(n^2) + O(n^2 \log n) + O(m) = O(n^2 \log n)$

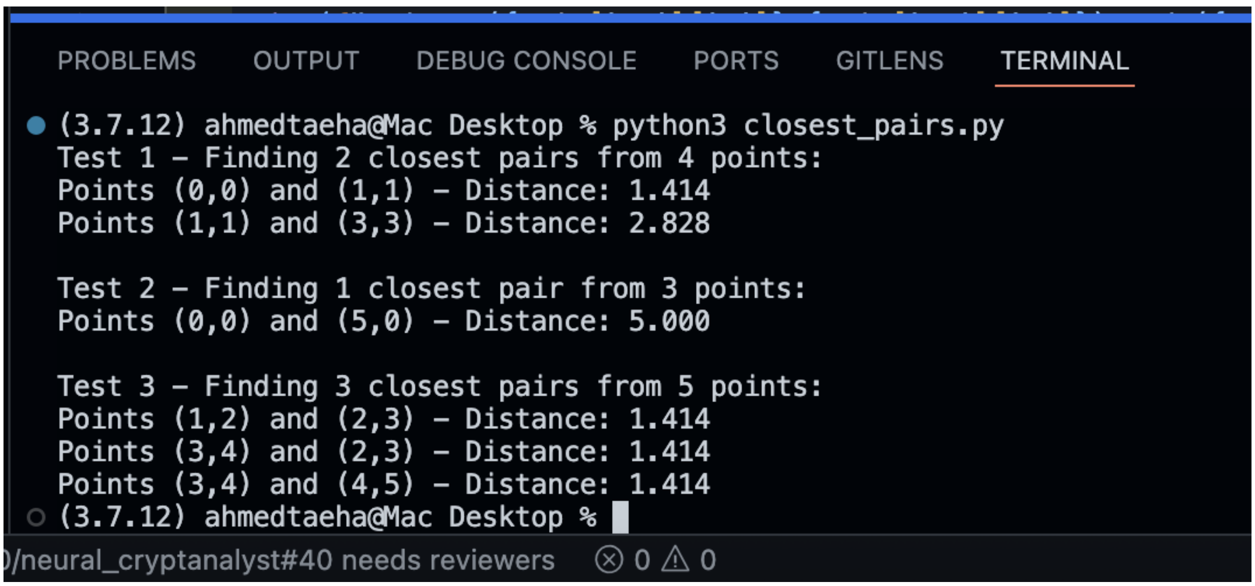
2(b).

```
import math

def find_m_closest(points_set, num_pairs):
    total_points = len(points_set)
```

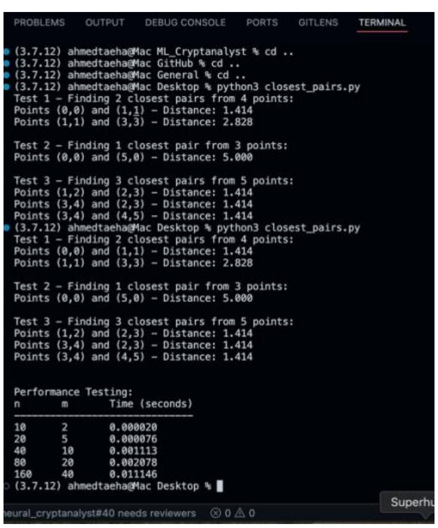
```
    pair_distances = []

    first_idx = 0
    while first_idx < total_points - 1:
        second_idx = first_idx + 1
        while second_idx < total_points:
            x_diff = points_set[first_idx]['x'] - points_set[second_idx]['x']
            y_diff = points_set[first_idx]['y'] - points_set[second_idx]['y']
            euclidean_dist = math.sqrt(x_diff * x_diff + y_diff * y_diff)
            pair_info = {
                'dist': euclidean_dist,
                'p1': points_set[first_idx],
                'p2': points_set[second_idx]
            }
            pair_distances.append(pair_info)
            second_idx = second_idx + 1
        first_idx = first_idx + 1

    pair_distances.sort(key=lambda x: x['dist'])

    closest_pairs = []
    counter = 0
    while counter < num_pairs:
        closest_pairs.append(pair_distances[counter])
        counter = counter + 1

    return closest_pairs

# Test Case 1: Simple 4 points
test1_points = [
    {'x': 0, 'y': 0},
    {'x': 1, 'y': 1},
    {'x': 3, 'y': 3},
    {'x': 5, 'y': 5}
]
result1 = find_m_closest(test1_points, 2)
print("Test 1 - Finding 2 closest pairs from 4 points:")
for pair in result1:
    print(f"Points ({pair['p1']['x']},{pair['p1']['y']}) and
({pair['p2']['x']},{pair['p2']['y']}) - Distance: {pair['dist']:.3f}")

# Test Case 2: Edge case with m = 1
test2_points = [
    {'x': 0, 'y': 0},
    {'x': 10, 'y': 0},
    {'x': 5, 'y': 0}
]
result2 = find_m_closest(test2_points, 1)
print("\nTest 2 - Finding 1 closest pair from 3 points:")
for pair in result2:
    print(f"Points ({pair['p1']['x']},{pair['p1']['y']}) and
({pair['p2']['x']},{pair['p2']['y']}) - Distance: {pair['dist']:.3f}")

# Test Case 3: Larger set
test3_points = [
    {'x': 1, 'y': 2},
    {'x': 3, 'y': 4},
```

```
    {'x': 5, 'y': 1},
    {'x': 2, 'y': 3},
    {'x': 4, 'y': 5}
]
result3 = find_m_closest(test3_points, 3)
print("\nTest 3 - Finding 3 closest pairs from 5 points:")
for pair in result3:
    print(f"Points ({pair['p1']['x']},{pair['p1']['y']}) and
({pair['p2']['x']},{pair['p2']['y']}) - Distance: {pair['dist']:.3f}")
```

2(c).



2(d).




```
import math
import time
```

```python
import random

def find_m_closest(points_set, num_pairs):
    total_points = len(points_set)
    pair_distances = []

    first_idx = 0
    while first_idx < total_points - 1:
        second_idx = first_idx + 1
        while second_idx < total_points:
            x_diff = points_set[first_idx]['x'] - points_set[second_idx]['x']
            y_diff = points_set[first_idx]['y'] - points_set[second_idx]['y']
            euclidean_dist = math.sqrt(x_diff * x_diff + y_diff * y_diff)
            pair_info = {
                'dist': euclidean_dist,
                'p1': points_set[first_idx],
                'p2': points_set[second_idx]
            }
            pair_distances.append(pair_info)
            second_idx = second_idx + 1
        first_idx = first_idx + 1

    pair_distances.sort(key=lambda x: x['dist'])

    closest_pairs = []
    counter = 0
    while counter < num_pairs:
        closest_pairs.append(pair_distances[counter])
        counter = counter + 1

    return closest_pairs

# Test Case 1: Simple 4 points
test1_points = [
    {'x': 0, 'y': 0},
    {'x': 1, 'y': 1},
    {'x': 3, 'y': 3},
    {'x': 5, 'y': 5}
]
result1 = find_m_closest(test1_points, 2)
print("Test 1 - Finding 2 closest pairs from 4 points:")
for pair in result1:
    print(f"Points ({pair['p1']['x']},{pair['p1']['y']}) and
({pair['p2']['x']},{pair['p2']['y']}) - Distance: {pair['dist']:.3f}")

# Test Case 2: Edge case with m = 1
test2_points = [
    {'x': 0, 'y': 0},
    {'x': 10, 'y': 0},
    {'x': 5, 'y': 0}
]
result2 = find_m_closest(test2_points, 1)
print("\nTest 2 - Finding 1 closest pair from 3 points:")
for pair in result2:
    print(f"Points ({pair['p1']['x']},{pair['p1']['y']}) and
({pair['p2']['x']},{pair['p2']['y']}) - Distance: {pair['dist']:.3f}")
```

```
# Test Case 3: Larger set
test3_points = [
    {'x': 1, 'y': 2},
    {'x': 3, 'y': 4},
    {'x': 5, 'y': 1},
    {'x': 2, 'y': 3},
    {'x': 4, 'y': 5}
]
result3 = find_m_closest(test3_points, 3)
print("\nTest 3 - Finding 3 closest pairs from 5 points:")
for pair in result3:
    print(f"Points ({pair['p1']['x']},{pair['p1']['y']}) and
({pair['p2']['x']},{pair['p2']['y']}) - Distance: {pair['dist']:.3f}")

def generate_random_points(n):
    points = []
    for i in range(n):
        points.append({'x': random.uniform(0, 100), 'y': random.uniform(0,
100)})
    return points

def measure_performance():
    test_sizes = [10, 20, 40, 80, 160]

    print("\n\nPerformance Testing:")
    print("n\tm\tTime (seconds)")
    print("-" * 30)

    for n in test_sizes:
        points = generate_random_points(n)
        m = n // 4

        start_time = time.time()
        find_m_closest(points, m)
        end_time = time.time()

        elapsed_time = end_time - start_time
        print(f"{n}\t{m}\t{elapsed_time:.6f}")

measure_performance()
```

2(e).

From n=10 to n=20 (2x increase), time went from .000020s to .000076s (3.8x increase). From n=20 to n=40 (2x increase), time went from .000076s to .001113s (14.6x increase). From n=40 to n=80 (2x increase), time went from .001113s to .002078s (1.9x increase). From n=80 to n=160 (2x increase), time went from .002078s to .011146s (5.4x increase).

The reason the time ratios are different is because of relatively small input sizes and system-wide measurements, but on average it's clear that there's superlinear growth, as predicted by $O(n^2 \log n)$. When n doubles we expect time to increase by about $2^2 \times \log(2n)/\log(n) \approx 4\text{-}8$ times which holds true for most of the measurements derived above.

The measured performance suggests that $O(n^2 \log n)$ was the correct expectation for the empirical testing because of:

- Small input sizes leading constant factors to dominate
- Overhead from the systems and background processes
- Python's dynamic memory allocation

3.
Improvements to the worst-case running time:

1. Divide and conquer. Instead of computing every distance for all $n(n-1)/2$ pairs, we can use a "plane-sweep" algorithm that divides by x-coordinate first and computes the closest distance for the first half before computing for the other half and combining. This gives us a worst case of $O(n \log n)$.
2. Pair generation early finishing. If we're keeping track of m pairs, when we reach position m, we can keep track of the mth smallest distance we've computed so far to avoid computing the distance from this point onward for any pair where delta x or y > this value.
3. A min-heap of size m. When generating pairs to keep track of the m closest while generating, we need not sort distances once we've computed $O(n^2 \log n)$; we can just keep a min-heap of size m while generating so the sorting costs are $O(n^2 \log m)$.
4. Spatial data structures. With information of where coordinates are in space, we can create a k-d tree or quadtree to avoid computing each point with every other point; we can skip over pairs that are far enough away and only check those that are close-by.
5. Avoid square root. When computing distances, compare squares instead since the square-rooting function is monotonic and this will save time per distance calculation.

The best improvement would definitely be a divide-and-conquer algorithm to achieve $O(n \log n)$ instead of $O(n^2 \log n)$.